Lucas Boppre Niehues

Estudo e Criação de um Editor de Código Estruturado

Florianópolis - Santa Catarina

2013/1

Lucas Boppre Niehues

Estudo e Criação de um Editor de Código Estruturado

Trabalho de conclusão de curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Universidade Federal de Santa Catarina - UFSC

Departamento de Informática e Estatística

Orientador: Prof. Dr. Olinto José Varela Furtado

Florianópolis - Santa Catarina

2013/1

Resumo

Virtualmente todos os editores de código modernos para uso profissional em linguagens de propósito geral são orientados a linhas e caracteres. A alternativa, editores que trabalham diretamente com a árvore sintática, recebe pouquíssima atenção nesta área, apesar de possuir vantagens importantes, como controle automático da estrutura e maior eficiência na escrita.

O objetivo deste trabalho é estudar estes editores alternativos, chamados editores estruturados ou editores dirigidos a sintaxe, comparando-os com os editores tradicionais. Uma vez levantadas as vantagens e desvantagens, será desenvolvido um editor estruturado que melhor se encaixe neste meio.

Este trabalho prevê primeiro um estudo aprofundado destes editores estruturados, no contexto da programação em linguagens de propósito geral ou não. Serão buscados esforços anteriores nesta área, avaliando seu impacto, lições aprendidas e limitações encontradas.

Em um segundo momento estas lições serão usadas para o desenvolvimento de um editor próprio, completamente estruturado ou híbrido, a fim de aumentar a eficiência do programador. Preferencialmente este editor estará preparado para uma linguagem de propósito geral moderna e será capaz de editar programas reais.

Espera-se deste trabalho uma melhor compreensão dos princípios por trás dos editores estruturados, buscando informações sobre sua eficiência, vantagens, desvantagens e fatores envolvidos na sua falta de adoção. Prevê-se também a construção de um editor estruturado funcional, capaz de ser usado no dia a dia, que faça melhor uso destas vantagens.

Lista de ilustrações

Sumário

Introdução

O editor de código é uma das ferramentas mais importantes para qualquer programador. É através dele que o programador lê, escreve e modifica seus programas, e portanto tem impacto direto na eficiência do profissional. Em vista disso, esta área sempre esteve em constante atividade, com milhares ou dezenas de milhares de editores já criados.

Naturalmente existe bastante variação nestas implementações, especialmente em relação ao nível de conhecimento da linguagem alvo, integração com outras ferramentas, qualidade das sugestões oferecidas e suporte a ações de refatoração. Apesar disto, uma variação específica é raramente considerada: editores estruturados, também chamados de editores dirigidos a sintaxe.

Um editor comum trata o código fonte como uma lista de linhas e caracteres. Ao usuário cabe inserir e remover linhas e caracteres a fim de escrever o código desejado. Esta abordagem é bastante simples, de fácil entendimento e uso, mas possui profundas limitações por causa do tipo de texto que está sendo escrito.

A sintaxe de virtualmente qualquer linguagem de propósito geral é composta por estruturas bem definidas que precisam obrigatoriamente seguir regras de construção e colocação. Por exemplo, a declaração de um método só pode ocorrer dentro do contexto de uma classe, e todo método deve possuir um segmento de código delimitado por chaves { }.

Em um editor comum o uso e aplicação correta destas regras recai completamente sobre o programador, que recebe ajuda somente pela coloração de sintaxe. Além disso os programadores também se submetem a regras adicionais, suas convenções de programação, que definem restrições como formatação desejada (nível de tabulação, posição dos caracteres de delimitação, espaçamento), nomenclatura de identificadores (ALL_CAPS, camelCase, under_score), organização de expressões booleanas e colocação de comentários.

É possível imaginar um editor de código onde todas estas regras fizessem parte do processo de edição, automaticamente produzindo apenas programas conformantes. Para isto basta lembrar que o código fonte é convertido para uma árvore sintática durante o processo de compilação ou interpretação, e este tipo de árvore possui as garantias desejadas.

Assim chega-se no conceito de editor estruturado, também conhecido como editor dirigido a sintaxe, que faz modificações diretamente na árvore sintática do código ao invés da sua representação textual. Este editor pode mostrar o código fonte de maneira textual,

mas dar ao usuário apenas comandos que modifiquem a árvore sintática. Um exemplo disso seria o comando inserir nodo, especificando que o tipo de nodo desejado é um bloco condicional if. Este comando garantiria que o bloco fosse esperado naquele local, que os delimitadores seriam inseridos corretamente e que a representação textual seguisse a formatação desejada pelo usuário. A cada passo do processo de edição o código fonte seria sintaticamente válido.

Outra vantagem dos editores estruturados é que fica mais fácil a análise do código fonte para gerar boas sugestões. Acesso ao namespace, com os tipos e variáveis disponíveis, torna trivial a função de auto completar, em contraste ao editor tradicional que depende de um compilador externo.

Por fim, este tipo de editor promove uma separação da visão do código e sua representação em disco. Esta separação permite que o código mostrado ao usuário possua modificações para facilitar a leitura e o entendimento. Um exemplo clássico é a omissão dos caracteres delimitadores de estruturas, como parênteses e chaves, que são redundantes quando a formatação está correta.

Naturalmente existem desvantagens com este conceito de edição de estrutura, especialmente quanto a limitação da linguagem alvo e interação com usuário. Qualquer editor estruturado precisa ter total conhecimento da linguagem alvo para garantir suas propriedades, o que restringe seu uso e dificulta seu desenvolvimento em linguagens complexas. A interação com o usuário também é outro sério problema, devido a dois fatores: comandos de edição e familiaridade.

Para um editor estruturado ser usado em ambientes reais, ele precisa permitir ao usuário uma edição eficiente do código fonte. Este problema cai completamente na escolha dos comandos de edição, como inserir, selecionar e apagar nodos, e como expor estes comandos ao usuário. Editores tradicionais ganham certa eficiência nesta área por permitir que o usuário tome atalhos, temporariamente criando código inválido para facilitar sua manipulação. Competir com este tipo de operação é um dos grandes desafios dos editores estruturados e a área em que boa parte dos esforços de pesquisa serão destinados.

O outro problema enfrentado é a familiaridade dos usuários aos editores de código textuais. Eles são conceitualmente simples, similares a outras ferramentas conhecidas (notepad, Word, campos de entrada na web) e estão em uso a tempo o bastante para que todos os usuários tenham grande prática. Este problema aumenta a barreira de entrada para editores alternativos.

Estes são os desafios que este trabalho irá abordar.

Parte I

Pesquisa

1 Fundamentação Teórica

Linguagens de programação possuem características bastante diferentes das linguagens naturais usadas no dia a dia, especialmente na forma que elas são construídas. Enquanto linguagens naturais possuem inúmeras regras gramaticais e suas exceções, linguagens de programação devem possuir construções sintaticamente simples e semanticamente precisas, para serem executadas por máquinas. É a diferença entre as construções de linguagens de programação e linguagens naturais que permite que programas sejam editados de formas diferentes: textualmente ou estruturalmente.

1.1 Linguagens Formais

Linguagens de programação devem ser entendíveis por ambos computadores e humanos para ser de alguma utilidade. Para solucionar o problema de legibilidade humana costuma-se utilizar texto plano, que possui o efeito colateral de maior portabilidade. Para ser legível e compreensível por máquinas tais textos são escritos seguindo uma lógica estrita, composta de regras léxicas, sintáticas e semânticas. Estas escolhas são especialmente importantes para garantir um fácil processamento (HERKEN, 1995).

Regras léxicas são aquelas que definem o alfabeto da linguagem, restringindo os caracteres permitidos. Um analisador léxico é o programa responsável por ler um texto na linguagem de programação e abstrair o alfabeto, retornando uma lista das palavras (tokens) neste texto para uso na próxima etapa da interpretação.

A sintaxe é definida através de uma gramática formal, definindo rigidamente as estruturas permitidas. A vantagem desta forma de especificação é que ela pode ser convertida diretamente em programas que analisam o código fonte e agrupam os tokens gerados pelo analisador léxico em uma estrutura em árvore (HOPCROFT, 2003). Estes analisadores sintáticos são chamados de parsers, e as estruturas que eles geram são as árvores sintáticas abstratas (AST).

Por fim existem as regras semânticas, que dão significado para a estrutura gerada. Estas regras não possuem um padrão comum para serem expressas, como uma gramática para as regras sintáticas, e portanto variam de linguagem para linguagem. Em linguagens de programação elas incluem restrições de tipos e escopo, por exemplo.

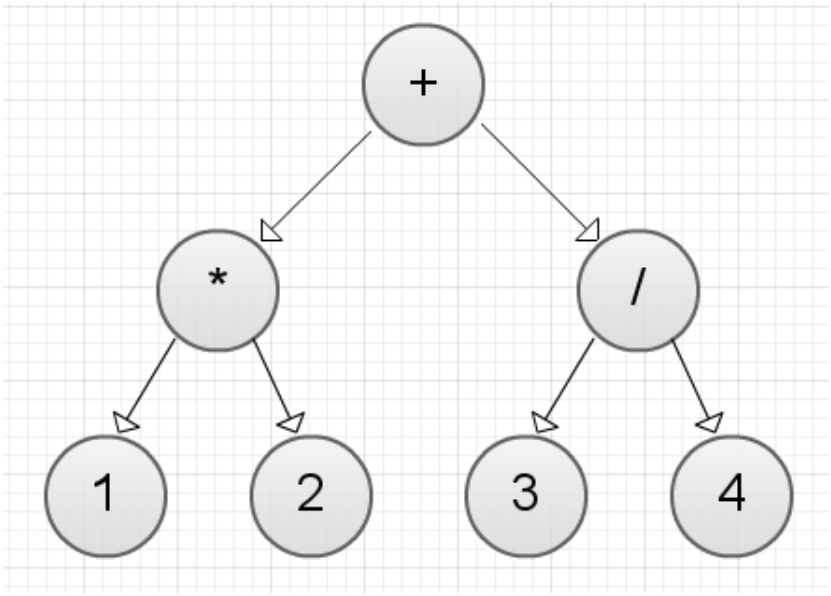

Figura 1 – Árvore sintática abstrata da expressão matemática (1 * 2) + (3 / 4)

1.2 Árvore Sintática

A análise sintática gera uma árvore com a estrutura do texto, onde cada agrupamento é representado por um nodo. Um exemplo de árvore sintática é a Figura 1, mostrando a árvore de uma expressão matemática.

Esta estrutura é equivalente ao texto original, mas é de muito mais fácil manipulação. Operações nesta estrutura costumam ser recursivas, permitindo que sejam executados sobre a estrutura inteira sem conhecimentos profundos da topologia. Por exemplo, para verificar a presença de um tipo de estrutura, basta programar esta verificação para uma estrutura e repetir o algoritmo para seus filhos. O resultado é que toda a árvore é analisada de maneira simples e elegante.

1.3 Linguagens de Programação

Linguagens de programação são linguagens formais, e portanto os conceitos acima podem ser aplicados à elas. As gramáticas, por exemplo, são usadas para especificar a sintaxe da linguagem, permitindo a criação de uma documentação não ambígua. As árvores sintáticas são usadas por compiladores para geração de código, e por analisadores de código para realizar verificações. Esta característica permite que ferramentas que operem em linguagens de programação reusem código e utilizem técnicas já bastante maduras para a análise sintática.

1.4 Texto x Estrutura

Por fim, a natureza das linguagens de programação permite dois paradigmas de edição: textual ou estrutural. Embora logicamente equivalentes, esta escolha define a dificuldade de implementar as funcionalidades. O realce correto de sintaxe, por exemplo, é trivial de ser realizado em um editor estruturado, enquanto a mudança de uma estrutura If para While é mais simples em editores textuais.

2 Funcionalidades dos Editores Estruturados

A partir do momento que o editor é capaz de manipular o código como uma estrutura de dados, várias novas funcionalidades se tornam disponíveis, algumas das quais serão citadas nesta seção. Estas funcionalidades variam em dificuldade para implementar e complexidade para usar, mas invariavelmente elas exigem algum grau de estruturação no código.

A importância destas funcionalidades varia muito em função dos objetivos e do contexto considerado. Por exemplo, enquanto a inclusão de um seletor de cores provavelmente não será uma grande impacto, a criação de macros locais e sugestões estruturais se destacam como promissoras. Mas vale lembrar que todos estes recursos modificam a experiência do usuário e exigem alguma interação, necessitando assim de maior atenção do usuário. Portanto a escolha de quais funcionalidades implementar e exibir para o usuário deve ser feita com cautela, a fim de evitar uma ferramenta que nenhum programador é capaz de dominar.

2.1 Organização de Código

Uma vez que os arquivos com códigos fontes foram lidos e interpretados, editores estruturados tem total liberdade de tratar este código da forma que for necessária. Esta possibilidade abre novos recursos, como utilizar diferentes unidades de código (mostrar métodos ou blocos ao invés de arquivos) ou exibir estas unidades de forma mais inteligente.

2.1.1 Diferentes unidades de código

Editores de texto usam o arquivo como unidade básica de código fonte. Arquivos são abertos, exibidos, editados e salvos. Esta é uma escolha natural dado o formato do programa em disco, mas limita severamente as formas do programador de manipular o código fonte.

Uma alternativa é tratar o código como grupos de algum elemento sintático, usualmente classes ou métodos. Smalltalk (SMALLTALK,) e, mais recentemente, Light Table (GRANGER, 2012), se destacam como interfaces que agrupam o código em métodos. O resultado é que o usuário tem um controle mais fino sobre o que exibir em sua tela. Pode parecer uma distinção trivial, mas navegação consome bastante do tempo de programadores e a escolha de um modelo mais natural diminui o esforço necessário para programar.

2.1.2 Exibir unidades por proximidade referencial e não proximidade declarativa

Virtualmente todos os editores textuais exibem as estruturas do código fonte, como métodos de uma classe, na ordem que elas foram escritas no arquivo. Programadores são cientes desta característica e ordenam as estruturas para fácil compreensão, mas nem sempre esta ordenação é eficiente para todos os contextos.

Editores estruturados são capazes de exibir estas estruturas de forma mais coerente, por exemplo agrupando métodos que se referenciam, ou exibindo classes na ordem de sua hierarquia de herança. Isto permite que o ambiente reflita de forma mais natural o contexto da tarefa, com ordem e agrupamento semântico.

2.2 Controles Embutidos no Código Fonte

Uma vez que o código fonte deixa de ser puramente textual, é possível embutir controles interativos nele para facilitar entrada de novo código. Estes recursos removem alguns problemas das linguagens de programação modernas como falta de clareza na unidade de medida e variações possíveis de parâmetros.

Infelizmente identificar o tipo de parâmetro e exibir o controle correspondente pode ser complicado. Alternativas incluem análises heurísticas (e.g. “isto se parece com um caminho de arquivo?”, “o nome deste parâmetro inclui a palavra color?”, “o texto da documentação menciona alguma unidade de medida?”) e anotação através de comentários. Se o editor não for limitado a tratamento de arquivos de texto, é possível também adicionar estas anotações em formato próprio durante o desenvolvimento do código.

2.2.1 Navegador de arquivos

Para entrada de caminhos de arquivos é possível incluir um navegador de arquivos, similar ao presente na maioria dos sistemas operacionais (OSENKOV, 2007). Normalmente esta operação exige que o usuário troque o contexto para um outro programa, localize o arquivo a ser referenciado, copie seu caminho de alguma forma (talvez até mentalmente) e insira este caminho no código fonte.

Se o editor for capaz de detectar que uma certa string é um caminho de arquivo, pode-se fornecer ao usuário um navegador de arquivos embutido para que o usuário selecione os caminhos desejados de forma gráfica e interativa. Seleção de múltiplos arquivos também pode ser suportada, com a saída no formato de uma lista de strings ou uma única string utilizando um separador (“arquivo1; arquivo2; arquivo3”).

2.2.2 Seletor de cores

Entrada de cores é uma necessidade frequente em programas com interfaces gráficas e jogos. Usualmente estes valores são expressos em números hexadecimais com 6 dígitos RGB, ou 8 dígitos para RGBA. Felizmente este formato é bastante incomum em outros contextos, fornecendo uma boa heurística para detecção de parâmetros de cores.

Uma vez que a semântica de um valor for identificada como de cor, um seletor de cores pode ser oferecido para o usuário. A escolha do controle em si fica a critério do usuário ou do editor, desde que seja respeitado o formato de saída.

2.2.3 Interface gráfica para entrada de expressões regulares

Expressões regulares são famosas por sua sintaxe concisa e dificuldade de leitura. Uma alternativa para sua escrita é um controle que permita exibição e edição de expressões regulares. A Figura 2 mostra a interface de edição de expressões regulares no ambiente Larch (LARCH. . . ,), exibindo a expressão regular em formato estruturado para facilitar o entendimento.

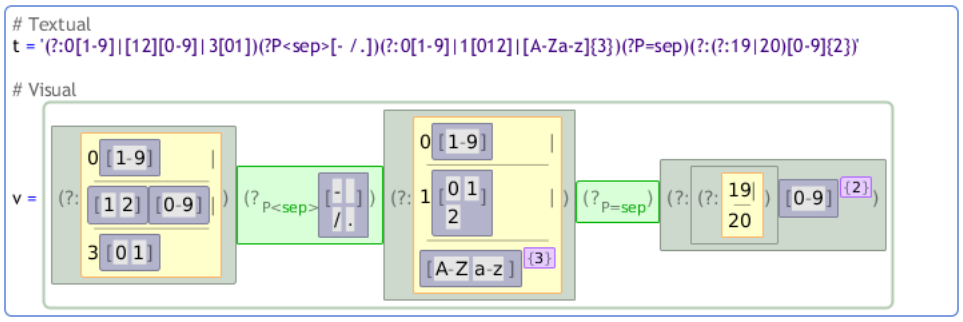

Figura 2 – Exibição de expressão regular com um controle embutido no ambiente de programação Larch

Fonte: http://larch.pythonanywhere.com/static/screenshots/visual_regex.png

2.2.4 Documentação com editor WYSIWYG

Documentação de classes e métodos em linguagens de programação modernas é feita através de comentários ou strings com formatos específicos. Existem vários formatos de documentação que são legíveis por máquinas, como Javadoc, cujo objetivo é que ferramentas externas possam entender os artefatos documentados, como seus parâmetros, exceções e valores retornados, para gerar páginas HTML ou PDF para publicação. Estes formatos são preferíveis por possuirem melhor formatação, suporte a hyperlinks e imagens. Naturalmente nenhum destes recursos está disponível nas strings e comentários de um código fonte em texto, mas isto se torna possível com um editor estruturado.

A entrada de documentação pode ser feita através de um controle WYSIWYG (What You See Is What You Get), que mostra a documentação escrita já com a formatação e elementos como apareceriam nas páginas publicadas. Este formato tem a vantagem de melhorar a legibilidade e permitir a inserção de ilustrações na documentação. Infelizmente este recurso costuma ocupar mais espaço que a alternativa de texto puro, necessitanto então de outras funcionalidades para ativar/desativar ou ocultar a documentação de métodos em edição.

2.2.5 Depurador e acesso a testes de unidade relacionados

Se o editor possuir integração com algum depurador para a linguagem, é possível inserir controles que contém a instância do depurador, com o programa em execução, dentro do ambiente de programação. Isto permite que o usuário compare execuções de versões diferentes do código e suas diferenças em comportamento através do depurador.

Também na área de testes estão os testes de unidade, usados para verificar o comportamento de trechos de código. Quando o programador editar estes trechos, os testes correspondentes podem ser exibidos, junto com informações sobre o resultado de suas execuções.

Juntas, estas funcionalidades trazem o comportamento real do código para mais próximo da fonte, permitindo a identificação de bugs mais facilmente e o melhor entendimento durante o processo de revisão. Algumas IDEs modernas já possuem suporte a estes recursos, mas são limitadas pela representação textual e a necessidade de sincronização do código em edição com a versão interpretada pelo compilador de fundo.

2.2.6 Visualizador de imagens e tocadores de vídeo e áudio para pré-visualização

Ao identificar um parâmetro que referencia um arquivo, uma opção do editor é permitir que o usuário visualize este arquivo. No caso de imagens, vídeo e áudio esta visualização pode ser através de um controle do tipo media player. Embora a edição destes conteúdos não seja simples, a sua visualização pode facilitar significativamente o entendimento do código.

Este controle pode ser estendido para visualizar outros artefatos, como um modelo 3D ou imagem aérea de um mapa no caso de um jogo. Se a API para acesso do editor for suficientemente madura e difundida, o próprio programa pode conter o código fonte deste controlador para melhor visualizar seus dados.

2.2.7 Console para execução instantânea

Um processo comum de desenvolvimento é escrever o código e rapidamente executá-lo para verificar que a saída segue o comportamento esperado. Normalmente este processo

conta com ferramentas como interpretadores da linguagem (REPL, read–eval–print loop), ou extrair o trecho de código sob avaliação para um arquivo temporário. Alguns editores, como o Light Table (GRANGER, 2012) possuem a funcionalidade de executar trechos pequenos de código individualmente, mas são necessárias áreas diferentes para exibir os resultados, adicionando ao número de elementos gráficos que o usuário tem que gerenciar.

2.2.8 Visualizador de documentação relacionada

A fim de diminuir a navegação necessária, um editor estruturado poderia extrair a documentação dos métodos em uso e exibi-la automaticamente conforme o usuário seleciona as chamadas de método (GRANGER, 2012). Isto permitiria que a documentação do método em uso esteja sempre visível, de forma passiva, incluindo lista de parâmetros e seus tipos esperados. Este tipo de funcionalidade é especialmente útil em linguagens dinamicamente tipadas, onde o tipo e até o número de parâmetros não é facilmente conhecido.

2.3 Simplificações

Uma importante vantagem dos editores estruturados é que o código exibido para o usuário não precisa ser idêntico ao que é salvo em arquivo. Seguindo o padrão MVC, a árvore sintática pode ser considerada o Modelo, enquanto o editor é o Controle e o código exibido a Visão.

A árvore sintática é serializada em duas situações diferentes: para ser exibida para o usuário e para ser escrita em disco. O resultado desta característica é que o código exibido para o usuário não precisa ser sintaticamente válido, podendo ser simplificado para melhor leitura. Novamente há um sacrifício a ser feito: embora estas simplificações facilitem a leitura, elas tornam o usuário menos proficiente na linguagem alvo. Esta desvantagem torna editores com este recurso inadequados para fins educativos, por exemplo, mas suas vantagens para uso comum mais do que compensam esta dificuldade.

2.3.1 Macros sintáticas, restritas ao editor atual

O usuário de um editor estruturado pode estender a sintaxe da sua linguagem com funções que simplifiquem o código mas não afetam o código gerado, similar a macros. Naturalmente estas macros são locais, salvas no editor do usuário. Embora pareça uma grande desvantagem, esta característica permite que cada programador crie suas macros preferidas sem necessitar de consenso geral ou criar conflitos entre programadores.

Macros sintáticas podem simular estruturas de outras linguagens, como switch-case em linguagens que não a possuem, e expressões comuns como “atribuir se nulo” (“a

= (a == null ? valor : a)"). Estas substituições podem ocorrer nas duas direções: converter uma macro em código quando o programa for salvo, ou converter um trecho de código lido em macros sintáticas quando o arquivo for carregado. Isto torna o recurso transparente para o usuário, efetivamente estendendo a linguagem.

2.3.2 Inserção ou remoção de parênteses em expressões matemáticas

Expressões matemáticas em linguagens de programação possuem precedência bem definida, mas nem sempre elas são claras ou conhecidas. Um usuário de editor estruturado pode solicitar que parênteses extras sejam inseridos em expressões com precedência desconhecida, ou removidos quando desnecessários. A vantagem deste recurso é que ele aumenta a legibilidade para este usuário e permite que a mudança seja salva no código fonte original, melhorando a legibilidade e possivelmente expondo bugs para o resto da equipe também, independente de editor utilizado.

2.3.3 Tratamento de identificadores como sentenças

Virtualmente todas as linguagens de programação modernas exigem que identificadores não contenham espaços. Como poucas variáveis e métodos podem ser descritos em apenas uma palavra, foram criadas convenções para converter sentenças em pedaços de texto que a linguagem considere como uma única palavra. Os principais métodos são camel-case (palavrasEmCaixaAlta) e underscore (palavras_entre_underscores).

A necessidade de se evitar espaços não existe num editor estruturado, uma vez que não há ambiguidade sintática se a propriedade "nome" de uma estrutura conter espaços. Um editor deste tipo pode então permitir ao usuário digitar sentenças normais, com palavras separadas por espaço para identificadores, variáveis, métodos e classes. Estas sentenças podem então ser "serializadas" ao formato desejado pelo usuário. O objetivo desta conversão é exibir os identificadores de uma forma mais legível para o usuário e remover a necessidade de cuidados especiais ao digitar identificadores.

Um problema acarretado por este recurso é a leitura destes identificadores, especialmente quando salvo em camel-case. Ao converter-se uma sentença para camel-case perde-se a informação das letras minúsculas e maíusculas, e portanto a conversão para este método é um processo "lossy", com possível perda de dados.

2.3.4 Omissão de símbolos desnecessários

Assim como a sintaxe pode ser estendida com macros, ela também pode ser reduzida. Se o editor garantir formatação correta, os símbolos de chaves podem ser seguramente removidos sem adicionar ambiguidade na maioria das linguagens. Nos casos de estruturas que contém outras, como if, for ou declaração de métodos, isto significa que os caracteres

que delimitam o início e fim do bloco podem ser omitidos, diminuindo o espaço necessário e número de caracteres a serem lidos.

Algumas linguagens modernas como Python utilizam indentação para delimitar blocos, a fim de melhorar a legibilidade (PYTHON...,). Outra otimização é a não obrigatoriedade dos parênteses ao redor de condicionais em blocos if e while. Com um editor estruturado estas escolhas ficam a critério do usuário, sem afetar nem mesmo outros programadores trabalhando no mesmo projeto.

2.3.5 Inserção e leitura de textos sem caracteres de escape ("\")

Um problema bastante comum nas linguagens de programação atuais é inserção de texto no código. Estas strings possuem delimitadores, normalmente aspas, e portanto qualquer ocorrência destes delimitadores dentro do texto deve ser tratada (ex: a = 'that's right', note as aspas usadas no texto). Em todos os editores encontrados este tratamento é completamente manual, exigindo cuidados do programador ao inserir ou copiar qualquer texto, tanto para dentro do código fonte quanto do código fonte para outros lugares. Outro problema são as strings de múltiplas linhas, usadas para textos literais que precisam incluir o caractere de quebra de linha. Algumas linguagens possuem uma tag para estes casos (ActionScript 3.0), ou usam uma combinação de aspas (Python).

Estes problemas são decorrentes dos textos literais possuírem o mesmo formato do código fonte ao redor, e é similar aos problemas que criam vulnerabilidades de segurança como XSS ou SQL Injection. Os problemas se agravam com textos mais complexos, como expressões regulares, que precisam utilizar o caractere contra barra para tratar alguns símbolos especiais. Como a contra barra já é utilizada pela linguagem hospedeira para escapar aspas e caracteres especiais como quebra de linha, esta contra barra precisa ser escapada novamente. Por exemplo, para uma expressão regular aceitar a string "*", a expressão na linguagem hospedeira pode exigir até seis contra-barras ("*").

Um editor estruturado evita o problema por não armazenar o código fonte como texto literal. Textos de uma ou mais linhas podem ter o mesmo tratamento e forma de inserção, exigindo apenas cuidado na hora de serializar o programa novamente para texto. Todo o tratamento de contra-barras, caracteres de nova linha e aspas pode ser realizado pelo editor de forma transparente ao usuário.

2.3.6 Formatação automática, tornar espaços em branco transparentes ao cursor

Ao mostrar o programa para o usuário, um editor estruturado possui a opção de exibir na formatação original (extraída durante o parsing) ou na formatação preferida do usuário. Praticamente todos os editores de código modernos possuem este recurso, mas a

diferença está no efeito colateral. Enquanto editores de texto precisam modificar o código fonte, e portanto alterar a formatação para todos os próximos programadores, um editor estruturado pode fazer isto de forma temporária, apenas para exibição ao usuário, assim como as macros sintáticas.

Este tipo de problema parece trivial, uma vez que é apenas uma decisão de onde colocar ou não caracteres de espaço em branco e símbolos opcionais. Na realidade este problema surge constantemente durante o trabalho de um programador e leva a discussões bastante sérias, uma vez que o padrão decidido terá que ser seguido por todos os programadores que alterarem o código fonte, além das trocas de formatação aparecerem nos sistemas de controle de versão (ATWOOD, 2009). Tornando a escolha de formatação pessoal a cada programador, estes problemas desaparecem e remove-se mais uma distração do profissional.

2.3.7 Eliminação de alguns erros sintáticos e semânticos

Naturalmente um editor capaz apenas de manipular estruturas válidas de uma linguagem é incapaz de produzir estruturas inválidas. Dependendo da implementação específica do editor, esta característica pode eliminar completamente a ocorrência de erros sintáticos durante a programação. Esta provavelmente é uma das propriedades mais conhecidas dos editores estruturados e suas consequências, como inabilidade de usar áreas de “rascunho”, também são conhecidas.

O que é pouco explorado é a anulação de algumas classes de erros semânticos. Se o editor tratar identificadores como ponteiros para as estruturas onde as variáveis / métodos / classes foram declarados, torna-se bastante difícil ou impossível utilizar variáveis não declaradas. Naturalmente uma restrição desta importância exige suporte especial, como sistemas de auto-complete agressivos e declaração de variáveis de forma extremamente rápida.

Outro problema semântico que pode ser evitado é a passagem de parâmetros do tipo incorreto. Dependendo da forma que os tipos de uma linguagem são declarados esta propriedade pode se restringir a passagem de valores literais do tipo errado (e.g. Python, onde os tipos das variáveis não são conhecidos em tempo de compilação), ou permitir a validação do tipo de todos os parâmetros (e.g. Java, onde o tipo das expressões é conhecido em tempo de compilação). O sistema de auto-complete mencionado anteriormente poderia se restringir a sugerir variáveis do tipo correto para aquele local, ou exibir apenas os métodos com o tipo de retorno compatível com a expressão ao redor.

2.4 Entendimento de alto nível

Uma vez que o editor possui conhecimento das estruturas que compõem o programa, sugestões de mais alto nível podem ser realizadas. Este tipo de recurso é especialmente útil de ser estendido através de plugins, adicionando funcionalidades inteligentes ao editor.

Para ser capaz de sugerir mudanças, o editor precisa ter acesso a um grande banco de dados de templates de estruturas e suas informações de desempenho, complexidade e outras características que se deseja analisar. Uma possível fonte destes dados são os projetos open-source e seus sistemas de controle de versão, contendo vastas quantidades de informação de utilização de estruturas (no próprio código fonte) e suas características (nos comentários do sistema de controle de versão e bug tracker). A mineração destes dados sem dúvida seria uma tarefa hérculea, mas é uma nova possibilidade aberta através da presença de editores estruturados.

Outra forma de criar este banco de dados é através de esforços voluntários. Se um editor estruturado obter grande adoção, os próprios usuários poderiam contribuir com estas informações. Esta abordagem seria especialmente útil no caso de estruturas comuns e com características bem conhecidas.

2.4.1 Sugestão de otimizações, simplificações ou expressões idiomáticas

Se o editor tiver acesso à informação de estruturas comuns e suas características (e.g. complexidade, desempenho), ele poderia ser capaz de sugerir mudanças no código criado. Estas sugestões podem incluir mudanças para melhorar o desempenho do código, simplificar sua estrutura ou utilizar expressões idiomáticas, mais comuns naquela linguagem. Este tipo de avaliação usualmente requer intervenção humana especializada no processo conhecido como code review, revisão de código. Ao tornar o editor capaz de realizar estas avaliações, a aprendizagem e qualidade de software aumenta para todos os usuários.

2.4.2 Predição de estruturas

Assim como editores textuais oferecem recursos de auto-complete para identificadores, um editor estruturado seria capaz de oferecer sugestões de prováveis estruturas. Ao identificar um trecho de estrutura conhecida, o editor forneceria para o usuário a opção de copiar o resto do padrão automaticamente. Esta alternativa diminuiria o esforço necessário para escrita de código e a chance de cometer erros durante esta escrita.

2.4.3 Sugestão de macros locais

Se o editor suportar macros sintáticas locais, é possível detectar padrões repetidos no código e sugerir ao usuário a criação de uma macro para contê-los. Naturalmente o processo da criação de macros envolve dois passos: identificar o padrão a ser abstraído e decidir a sua substituição. A menos que o editor tenha acesso a uma base de macros comuns, ele seria capaz apenas de realizar sugestões para o primeiro passo, a identificação do padrão. A criação de uma estrutura substituta continuaria a cargo do usuário.

2.5 Novas ferramentas

A existência de editores estruturados permite criar algumas novas ferramentas, independentes das funcionalidades de leitura e escrita citadas acima. Embora todas as aplicações futuras não possam ser previstas, é possível imaginar algumas que fariam melhorias significativas no trabalho de seus usuários, como implementação trivial de refatorações, busca e comparação semântica.

2.5.1 Refatoração trivial

Muitas IDEs modernas possuem funcionalidades de refatoração de código, como renomear variáveis, extrair classes pai ou isolar expressões em novas variáveis. Porém estas operações precisam ser feitas com extremo cuidado, da forma mais conservativa possível, pois manipulações textuais erradas podem facilmente criar erros sintáticos ou semânticos. Outra característica é que estas operações costumam ser lentas pois exigem muito processamento sobre código fonte original.

Um recurso oferecido pelos editores estruturados é a refatoração simplificada. Mover estruturas se torna uma operação com tempo de execução constante e bastante trivial de se realizar. Renomeação de variáveis pode ser o comportamento padrão ao mudar o identificador utilizado em uma expressão, novamente sem impacto no desempenho do editor. Uma vez que estas operações se tornem mais simples, talvez novas operações sejam construídas, exclusivas para editores estruturados, como movimentação de blocos de código ou extração de expressões comuns no código fonte.

2.5.2 Busca sintática

Ocasionalmente um programador precisa buscar variáveis, métodos ou classes com um certo nome ou uma certa característica. Algumas IDEs textuais implementam esta funcionalidade com compiladores de fundo, mas estas estruturas não são acessíveis diretamente pelo programador e podem estar desatualizadas.

Uma busca sintática permitiria ao usuário especificar um padrão a ser encontrado no código fonte. Este padrão poderia ser composto com palavras chaves e tipos, como "métodos com a palavra "add" no nome", ou especificação completa de estruturas, como "blocos condicionais contendo instanciação de objetos". A interface para especificação destes padrões pode ser similar a do próprio editor, com uma expressão extra para identificar valores irrelevantes a busca. O objetivo de um recurso de busca desta natureza é permitir que o usuário encontre trechos de código que precisam de correção ou que possam ser fontes de problemas, como uso de métodos inseguros ou estruturas não otimizadas.

2.5.3 Comparação de versões de forma semântica

Ferramentas atuais de controle de versão fazem comparação por linha, tornando difícil entender quando um método foi criado ou movido de outro lugar, por exemplo. Questões mais complexas, como "em que classe este método foi originalmente criado?" são praticamente impossíveis de responder com os recursos atuais. Editores estruturados vêem o código em um nível mais alto, permitindo que este tipo de sentido seja extraído das próprias ações do usuário.

Se o editor descrever as alterações feitas no código, do seu ponto de vista estruturado, a semântica necessária para responder àquela questão se torna disponível. Assim como o recurso de busca sintática, o objetivo desta funcionalidade é identificar possíveis problemas e melhor entender o código do programa. Se o programador notar que métodos movidos de uma classe para outra costumam apresentar problemas, um controle de versões com informações semânticas seria capaz de encontrar estas instâncias.

Naturalmente este recurso faz várias exigências importantes. A primeira é que haja comunicação entre o sistema de controle de versão e o editor, o que no mínimo exige uma API e retro-compatiblidade. A segunda exigência é que todos os programadores utilizem editores compatíveis para não deixar lacunas no histórico do código. E por fim, este sistema como um todo precisa de novo software e seria muito provavelmente incompatível com sistemas similares. Mesmo assim estes recursos podem encontrar alguma área onde as vantagens sejam mais importantes que estas exigências.

3 Editores Existentes

Esta seção descreve alguns editores de código existentes, estruturados ou apenas com características similares, e tem como objetivo identificar e avaliar os problemas naturais a estes editores, as soluções utilizadas e as funcionalidades únicas de cada um. Dada a tremenda diferença na abordagem dos problemas, esta avaliação deve ser feita de maneira individual para cada editor. Um fato notável é que uma busca profunda encontrou vários editores desta categoria, mas poucos com notabilidade suficiente, a maioria sendo de meios acadêmicos. Isto permite uma análise mais detalhada das propriedades descobertas, mas há pouca informação sobre o uso real destes editores.

É interessante notar que este assunto está ganhando novo vigor nos meses recentes, e prova disso é o Light Table e o Tiled Text, lançados durante o desenvolvimento deste trabalho. São projetos em fase alpha, ainda não usáveis no dia-a-dia, mas bastante elogiados pela comunidade técnica. O problema de criar um editor estruturado continua em aberto, mas estes projetos são prova do esforço revigorado na área.

3.1 Larch

Larch (2011, http://larch.pythonanywhere.com/) é um ambiente de programação Python que é capaz de lidar com controles gráficos. O desenvolvimento é feito em worksheets, páginas que contém tanto o código fonte quanto a saída do programa criado, e principalmente suporta objetos gráficos interativos no código fonte e na saída, como exemplificado na Figura 3.

Worksheets permitem incluir elementos gráficos e interativos no código, tornando o código mais legível e fácil de ser manipulado. Alguns exemplos de elementos gráficos são os consoles de depuração e substituição de matrizes por tabelas com constantes ou até trechos de código. Estes elementos são inseridos no código quando certos padrões de texto são identificados, como é o caso do editor de expressões regulares. Assim que o sistema identifica um padrão válido na expressão regular, ele é substituído por um bloco visual equivalente para facilitar a leitura.

3.2 Kodu

Kodu (2009, https://research.microsoft.com/en-us/projects/kodu/) é um ambiente de programação visual com foco em criação de jogos 3D simples utilizando uma linguagem visual. O ambiente foi desenvolvido pelo FUSE Labs da Microsoft e funciona

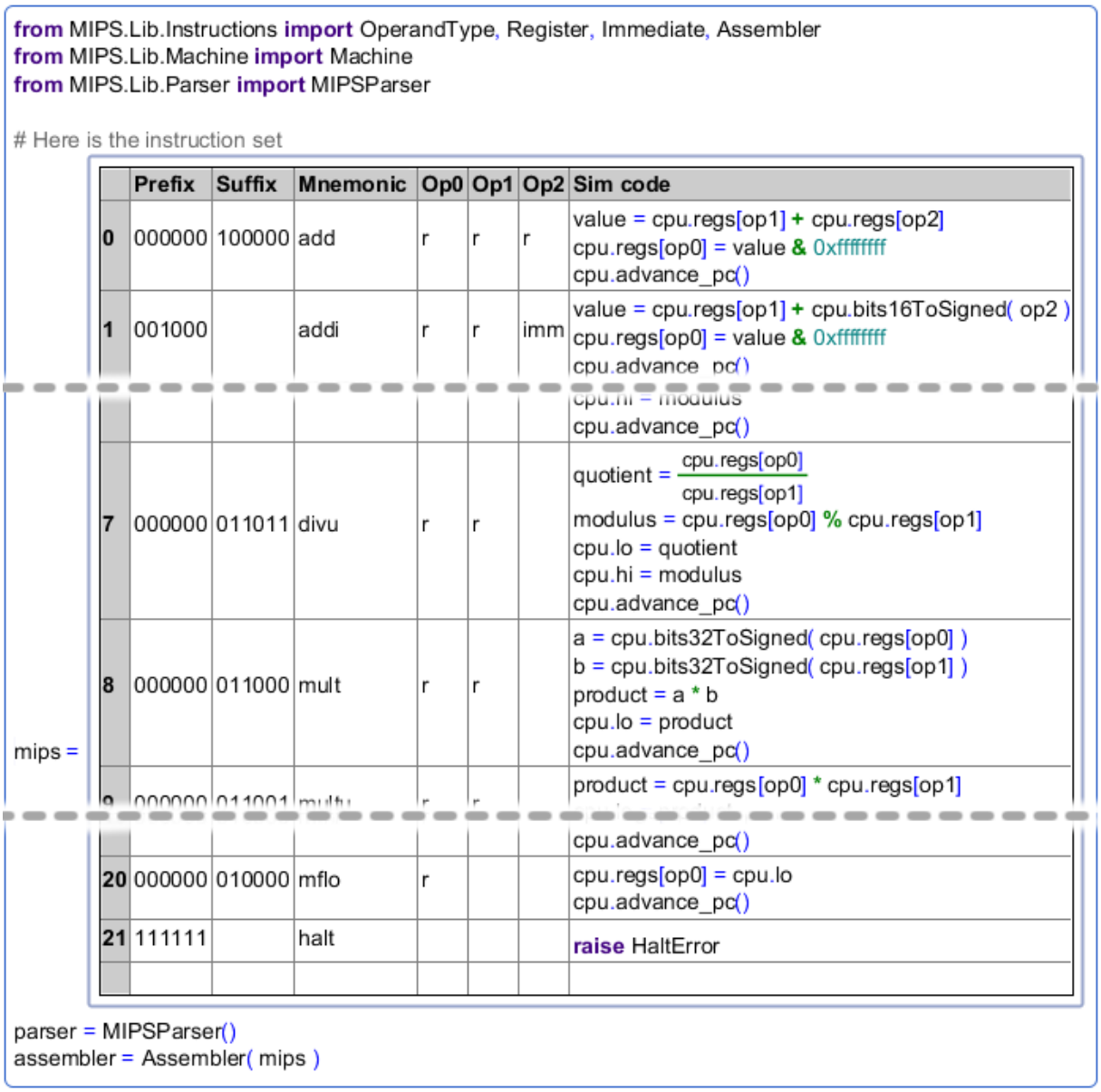

Figura 3 – Exemplo de código fonte mesclado com controles interativos no editor Larch

Fonte: http://larch.pythonanywhere.com/static/screenshots/mips_sim.png

em Xbox 360 e Windows, sendo jovens programadores seu público alvo. A interface do comportamento dos objetos pode ser vista na Figura 4.

O ambiente modela o processo comum de desenvolvimento: programação, teste e divulgação em interfaces diferentes. A programação do jogo é feita com pares condições / consequência, com os componentes escolhidos de uma lista pré-definida. Toda a interface é otimizada para controladores de video game, com os componentes selecionados de um círculo e o gerenciamento dos pares condição / consequência em linhas numeradas.

A sintaxe da linguagem é bastante simples, contendo apenas sentenças divididas entre condição e consequência. Cada seção desta pode conter módulos da lista pré-definida

Figura 4 – Interface do Kodu para definir o comportamento dos objetos

Fonte: http://research.microsoft.com/en-us/projects/kodu/programming_ui.jpg

e possivelmente especializações também pré-definidas (ex: "controlador é usado" pode ser especializado em "direcional esquerdo do controlador é usado"). O usuário utiliza o controlador para selecionar as partes a serem editadas, ou o símbolo de "+" ao final de cada seção para adicionar novas especificações.

3.3 Greenfoot

Greenfoot (2006, http://www.greenfoot.org/door) é uma IDE educativa para ensinar o conceito de orientação a objeto. Embora a declaração de métodos e suas implementações sejam escritas em um editor textual comum, a criação de classes e suas estruturas hierárquicas são feitas através de uma interface gráfica, como na Figura 5.

A declaração de novas classes e seus relacionamentos é feita de maneira completamente gráfica, utilizando menus e botões comuns. Por ser um ambiente educativo, também é possível declarar instâncias em posições no tabuleiro, que é uma forma de edição estruturada também, embora bastante limitada por ser capaz apenas de criar instâncias e definir suas posições. Os blocos de código como condicionais ou corpo de métodos são coloridos sintaticamente, auxiliando na leitura do programa.

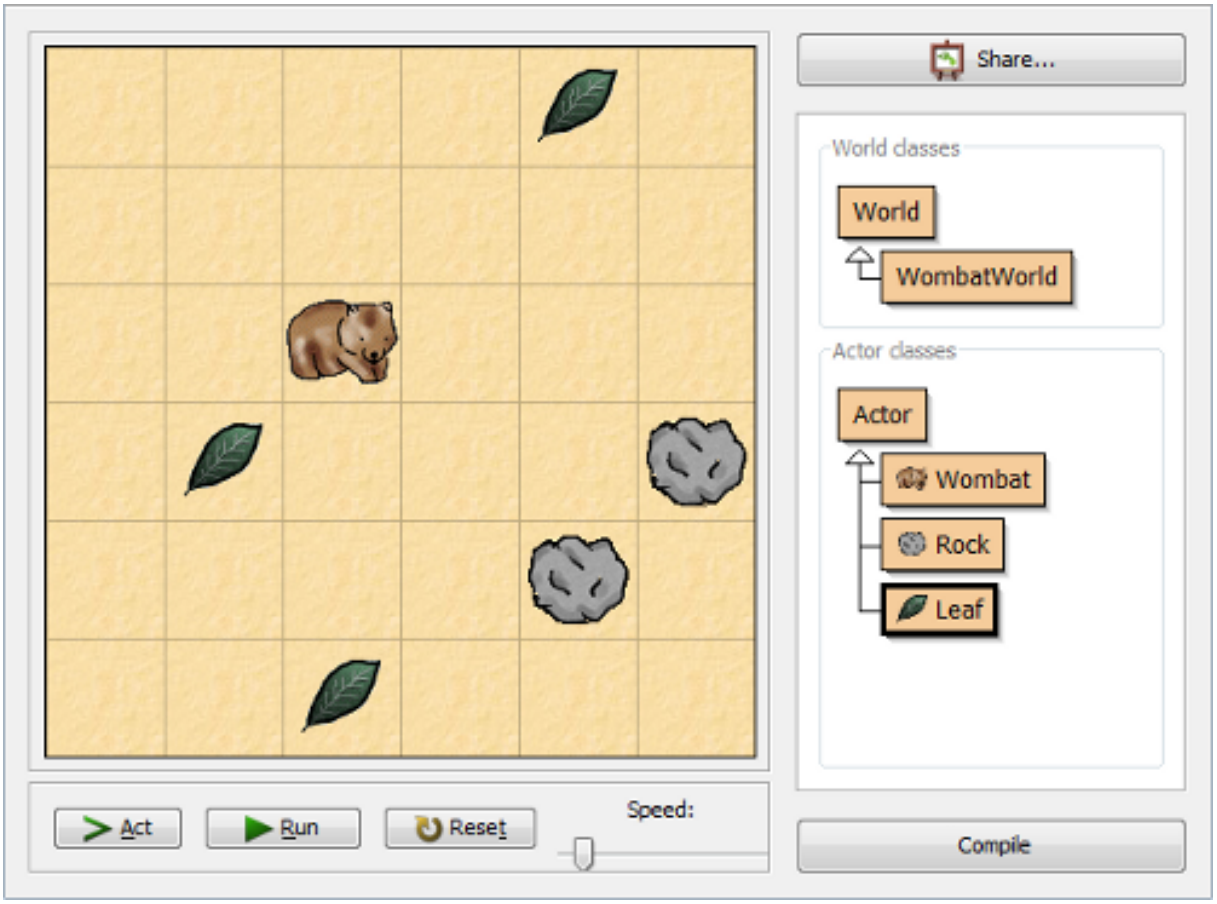

Figura 5 – Interface do Greenfoot para criação de instâncias e execução do programa

Fonte: http://www.greenfoot.org/images/overview/visual.png?1345204189

3.4 Code Bubbles

Code Bubbles (2010, http://www.andrewbragdon.com/codebubbles_site.asp) é um framework para criação de IDEs Java, com implementações para Eclipse e Visual Studio. O objetivo é fornecer um ambiente de trabalho para o programador em uma área 2D, com métodos, notas, documentação e textos em geral sendo contidos em bolhas móveis neste ambiente (Figura 6).

Ao contrário das IDEs normais, Code Bubbles trata métodos como a menor unidade de código, permitindo o usuário visualizar a implementação de vários métodos relacionados em uma mesma janela. O foco em métodos e não em classes, a presença de outros fragmentos como notas, documentação e até sessões interativas de depuração permitem que o programador tenha acesso a mais contexto e diminui a necessidade de navegação adicional.

O ambiente 2D também pode ser manipulado. Seções dele podem ser agrupadas e salvas para uso posterior, ou enviadas por email, podendo ser usadas para designar tare-

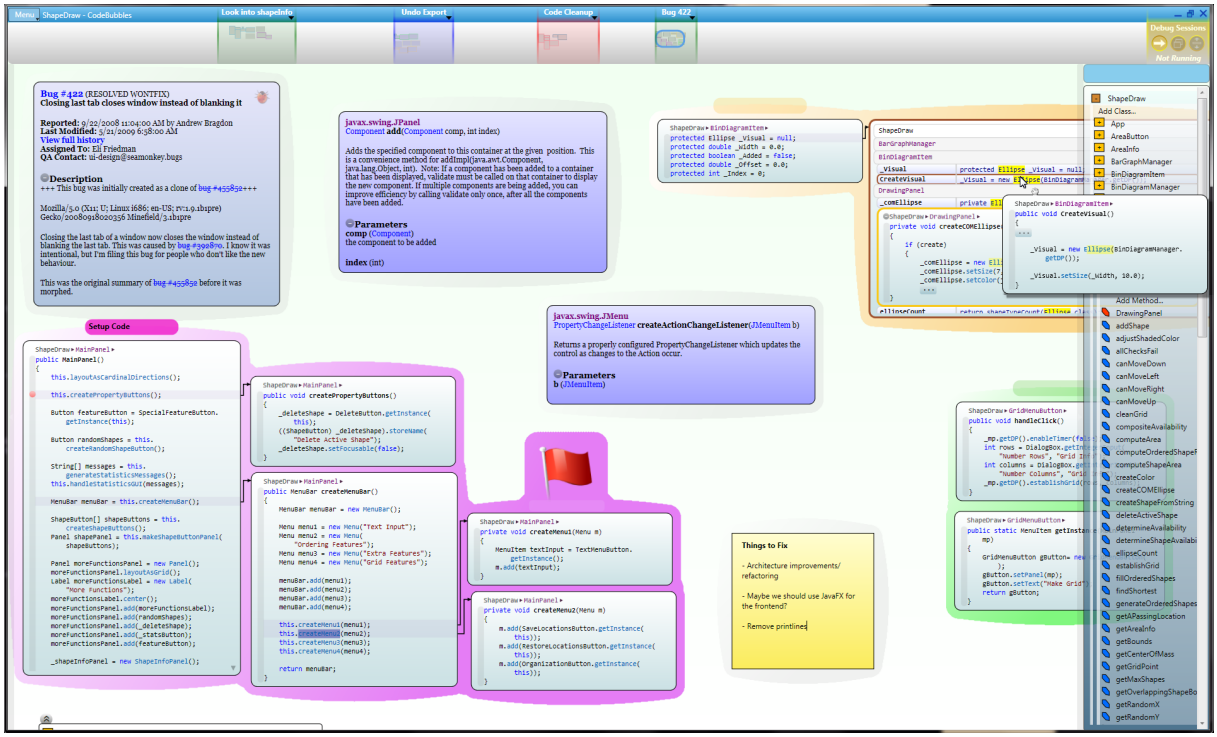

Figura 6 – Code Bubbles para visual studio, mostrando diversas “bolhas” com os recursos principais da IDE: documentação, métodos individuais, agrupamento de bolhas, seções

Fonte: http://www.andrewbragdon.com/screenshots/codebubbles_screenshot1.png

fas individuais. Há uma grande preocupação em manter as seções visualmente distintas, facilitando a navegação.

Embora os editores dentro das “bolhas” sejam orientados a texto, os níveis superiores são altamente estruturados. A manipulação dos métodos como estruturas semi-atômicas tornou possível a criação de um ambiente completamente diferente dos ambientes de programação tradicionais, capaz de reduzir o tempo e número de operações necessárias para completar tarefas.

3.5 Scratch

Scratch (2006, http://scratch.mit.edu/) é uma ferramenta de programação para educação infantil que inclui uma linguagem de programação própria e ambiente de desenvolvimento (Figura 7). A linguagem possui recursos para manipulação de imagens e sons, com suporte a interação, tornando a ferramenta bastante popular para criação de jogos.

A seção que descreve o comportamento do programa é escrita de forma completamente estruturada, utilizando uma analogia de blocos encaixados. A sintaxe do programa é descrita através das conexões nos blocos: comandos são representados por um bloco com

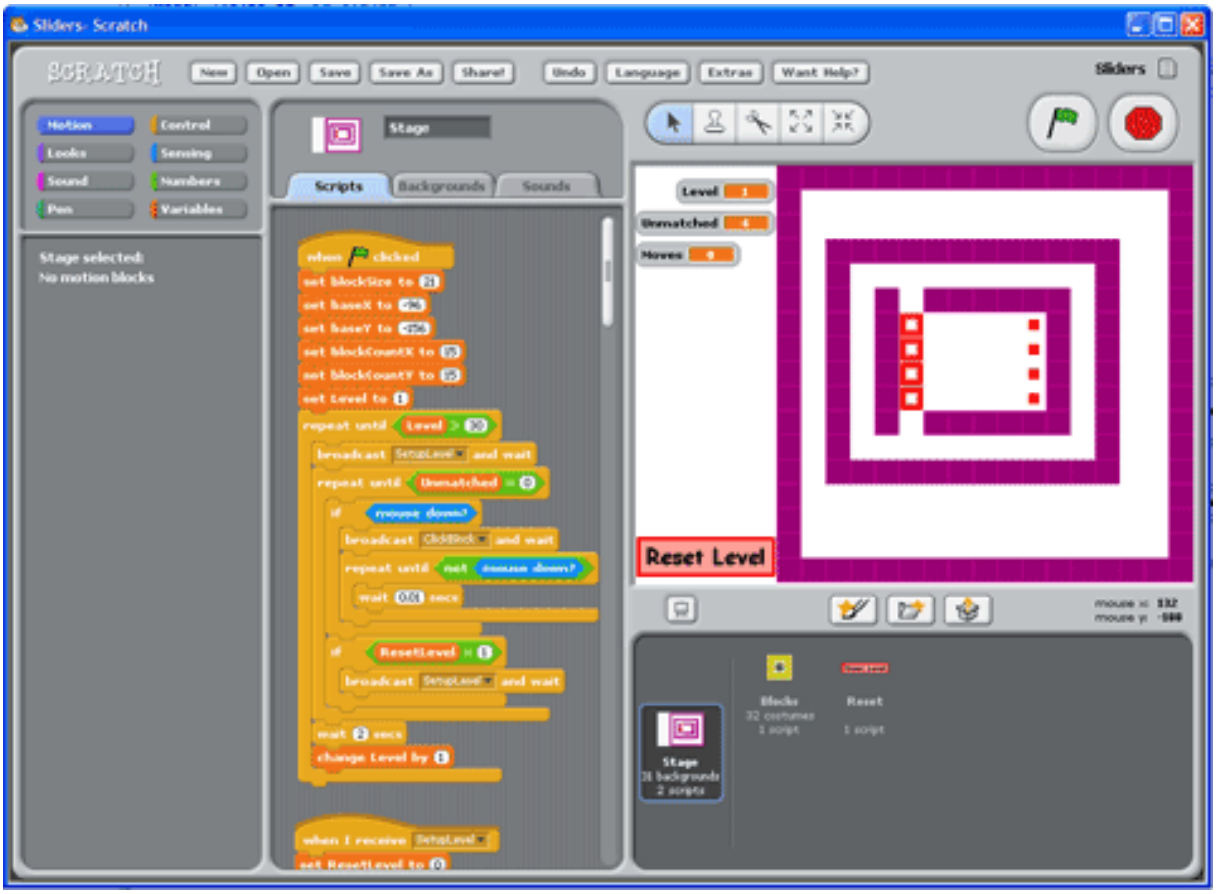

Figura 7 – Interface principal do Scratch, mostrando o código fonte a esquerda e o programa em execução a direita

Fonte: http://origin.arstechnica.com/journals/microsoft.media/scratch1.png

uma pequena fenda retangular na parte de cima e uma parte de formato correspondente na parte de baixo. Isto permite que estes blocos sejam colocados em sequência, um encaixado sobre o outro. Condicionais possuem um espaço para colocar blocos de comando, que se expande automaticamente, e uma fenda em formato de losango no espaço da condição, onde o usuário pode colocar blocos de expressão booleana.

Esta interface (Figura 8) possui diversas características interessantes. A primeira é que a sintaxe está sempre garantidamente correta, já que o usuário não é capaz de "quebrar" blocos, mas ainda há espaço para estados intermediários. Ao mover um ou mais blocos de comando para fora de um início de evento, é possível criar um pedaço de programa que jamais será executado, por não ter ponto de partida, mas pode ser usado como rascunho. O equivalente em uma linguagem de programação comum seria possuir um pedaço de código comentado ou dentro de uma função não utilizada.

A segunda característica é que ela é auto documentada, já que os blocos possuem suas ações escritas na parte sólida, permitindo ler o conteúdo dos blocos como um programa comum. Estas duas características juntas garantem uma fácil leitura e entendi-

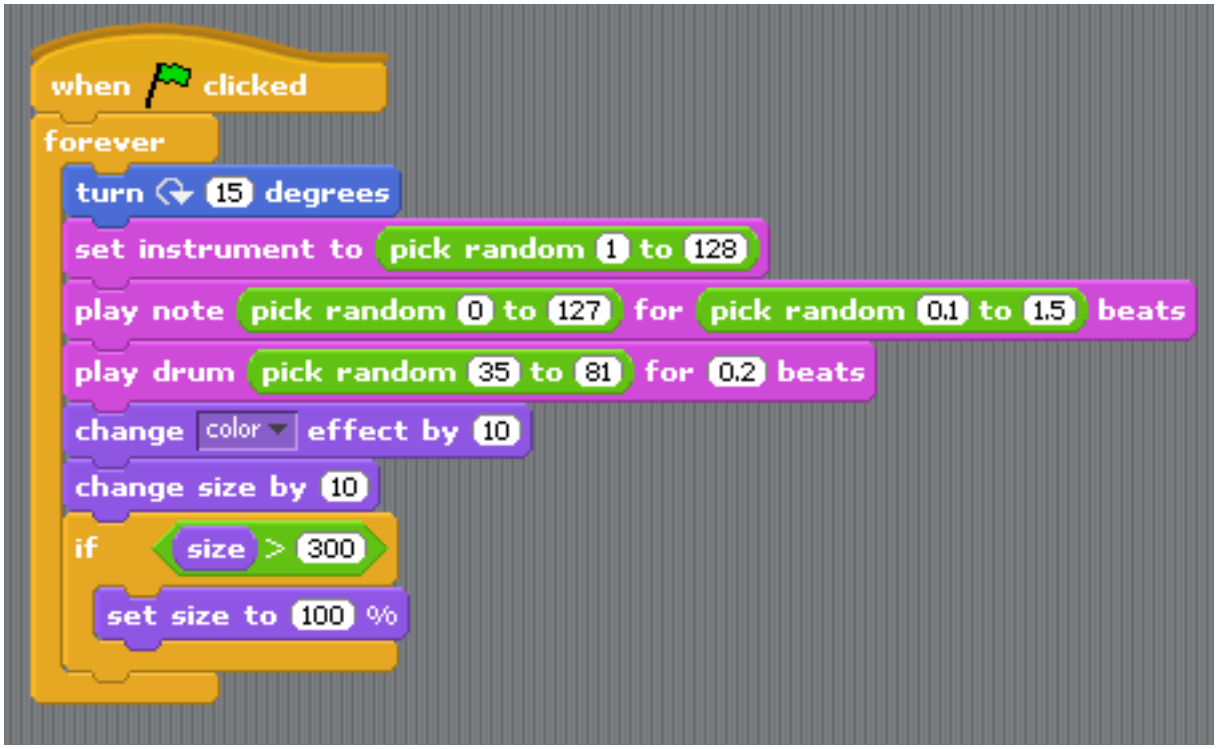

Figura 8 – Exemplo de programa escrito no Scratch. Os blocos se encaixam fisicamente uns nos outros, enquanto containers como forever e if esticam para conter os comandos contidos.

Fonte: http://blog.srinivasan.biz/wp-content/uploads/2008/04/scratch1.png

mento do programa, absolutamente necessário para programas educacionais como este.

Por último, esta interface é extremamente simples de aprender e fácil de descobrir novos recursos. Todos os blocos possíveis são acessíveis a partir de um menu simples, suas ações facilmente legíveis pelo texto contido neles, e o formato do bloco explicita o seu possível uso (condição, comando, evento inicial, container para comandos).

3.6 Light Table

Light Table (2012, http://www.chris-granger.com/lighttable/) é uma IDE para Clojure e Python que abraça customização do ambiente e possui vários recursos embutidos (Figura 9). Um dos recursos mais interessantes é o tratamento da documentação, que está sempre disponível para o método atualmente selecionado e é pesquisável por nome (e.g. pacote em que ela se encontra) ou busca textual (e.g. descrição do método). O ambiente também inclui um console com avaliação instantânea, capaz de exibir o código fonte com as variáveis substituídas por seus valores avaliados, facilitando o processo de depuração.

As unidades de manipulação do código são os métodos, que podem ser manipula-

Figura 9 – Exemplo de interface do Light Table, mostrando funções espalhadas pelo ambiente e um espaço para exibição do programa final

Fonte: http://www.chris-granger.com/images/lightable/game-example.png

dos e organizados individualmente. Algumas facilidades disponíveis para estas unidades são a avaliação instantânea, posicionamento arbitrário na interface e substituição das variáveis por seus valores durante depuração. A edição de código em Python possui algumas funcionalidades extras, como inclusão de resultados, erros e imagens embutidos no próprio texto do código.

3.7 Tiled Text

Tiled Text (2013, http://www.tiledtext.com/) é um editor experimental, ainda incompleto, para manipulação de código sem auxílio de mouse ou teclado. O foco é em dispositivos periféricos alternativos como telas sensíveis ao toque ou controles de videogame. O código é exibido na parte esquerda, com uma certa granularidade (neste caso em letras individuais) e uma árvore de derivação parcial é mostrada a direita, indicando quais nodos estão sendo selecionados (Figura 10).

Embora faltando muitas das funcionalidades requeridas para um editor de código, como inserção de novas estruturas, a versão atual possui muitas características novas que merecem ser avaliadas. A primeira é o controle do código em diferentes níveis de granu-

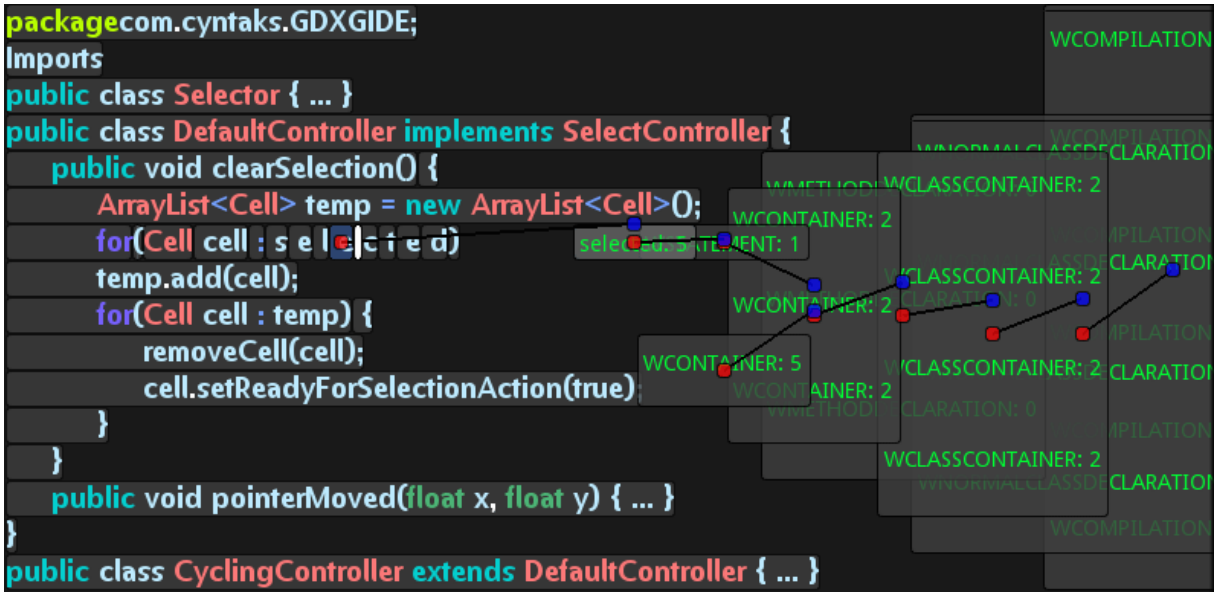

Figura 10 – Imagem do editor Tiled Text mostrando a funcionalidade de edição

Fonte: http://www.tiledtext.com/resources/banner2.png

laridade. O usuário é capaz de "subir" ou "descer" níveis no código, exibindo estruturas diferentes conforme o nível muda. No nível mais alto estão as declarações de classes, com suas implementações omitidas (public class Selector ... na imagem acima). Ao descer níveis o usuário pode selecionar métodos destas classes, linhas de código destes métodos, tokens destas linhas e por fim caracteres individuais destes tokens. Esta forma de navegação sacrifica um pouco da simplicidade dos editores convencionais para acelerar a navegação, já que o cursor pode ignorar seções que não são de interesse do usuário. Esta visão de alto nível também facilita o entendimento do código, pois mostra de forma compacta as classes do projeto e seus métodos.

Outra característica interessante deste editor é a movimentação de código. Em editores comuns o usuário pode recortar um trecho de código para a "área de transferência", tornando-o disponível para inserção em outros locais. O Tiled Text aborda o problema de uma forma diferente, "arrastando" o trecho de código junto com o cursor. Esta seleção é mostrada como um bloco de cor sólida acompanhando o cursor, movendo o código antes e depois da seleção para acomodar o novo trecho. Este recurso se aproveita do fato que o mesmo bloco de código raramente é copiado para mais de um destino, sacrificando uma das vantagens da área de transferência para atender o caso de uso mais comum. O bloco de seleção ocupa um espaço significativo na tela, mas a sua semântica é óbvia para o usuário e ajuda a prever a consequência de suas ações.

3.8 Tabela Comparativa

A fim de melhor entender e resumir o panorama de editores estruturados foi montada uma tabela comparando os principais editores desta área. As características analisadas foram:

a) **Linguagem**: linguagens suportadas pelo editor, ou se há suporte apenas a uma linguagem própria.

b) **Exibição**: se o código é exibido de forma completamente gráfica (todos os nodos são imagens isoladas), completamente textual (a árvore sintática é representada de forma similar ao código original) ou alguma forma híbrida.

c) **Interface**: ferramentas que o usuário utiliza para interagir com o editor. Na prática as únicas formas encontradas foram mouse, teclado e controles (gamepad).

d) **Tipo de editor**: se o editor em si se comporta de forma completamente estruturada (todas as edições devem ser na árvore sintática), completamente textual (todas as edições são no texto de origem) ou híbrida.

e) **Formato de arquivo**: qual o tipo de arquivo que o editor lê e escreve. Aqui dividiu-se os formatos em texto (compatíveis com outros editores) ou binário (formato próprio).

f) **Estende a linguagem**: se o editor contém recursos que não são nativos da linguagem utilizada. Os principais recursos de extensão encontrados foram controles embutidos no código fonte (tabelas, campos de texto, botões) e agrupamentos de métodos ou classes além daqueles disponíveis na linguagem. Naturalmente estas extensões exigem um formato de arquivo próprio ou são perdidas no ciclo de salvar/carregar o projeto.

	Linguagem	**Exibição**	**Interface**	**Tipo de editor**	**Formato arquivo**	**Estende a linguagem?**
Larch	Python	híbrida	mouse, teclado	híbrido	binário	sim
Kodu	própria	gráfica	controle	estruturado	binário	não
Greenfoot	Java	híbrida	mouse, teclado	híbrido	binário	não
Code Bubbles	Java	híbrida	mouse, teclado	híbrido	texto	sim
Scratch	própria	gráfica	mouse, teclado	estruturado	binário	não
Light Table	Clojure, Javascript, Python	híbrida	mouse, teclado	texto	binário	sim
Tiled Text	Java	texto	teclado, controle	estruturado	texto	não
Editor proposto	Flexível	texto	mouse, teclado	estruturado	texto	sim

4 Características de um Editor Estruturado

Algumas características de editores estruturados são obrigatórias, como movimentação do cursor e criação de novas estruturas. Estas características foram agrupadas em duas áreas gerais: navegação e edição. Estas áreas são significativamente diferentes dos editores textuais e usadas constantemente, exigindo cuidados especiais com usabilidade e desempenho.

Estas são as características que formam o núcleo do editor, independente de linguagem. As operações aqui descritas são baseadas em uma estrutura em árvore genérica, cuja única exigência é a capacidade de ser serializada para exibição. Esta é a interface principal do usuário com o editor, então aqui se encontram os maiores problemas com a falta de familiaridade, já que as funcionalidades conhecidas são substituídas completamente.

4.1 Navegação

Navegação se refere ao movimento do usuário ao longo do código, sendo um recurso orientado a leitura. Editores convencionais utilizam quatro funcionalidades principais: navegação local manual (mover uma linha/página para cima/baixo, pular para o início/fim), salto para linhas numeradas, salto para ocorrências de uma busca e divisão do código por arquivo, normalmente em abas. Todas estas funcionalidades apresentam desafios para editores estruturados, mas também existem novas funcionalidades que se tornam possíveis.

4.1.1 Numeração das linhas

Sem conhecimento prévio do código serializado em texto, o editor não é capaz de realizar navegação por linhas. Um comando para "ir para a linha 1160"necessitaria serializar todas as estruturas do programa, começando pela raiz, até exatamente 1160 linhas serem geradas.

Uma possível solução é manter uma referência, nas estruturas, das suas respectivas linhas no texto original. Modificações no código não seriam refletidas nestas referências, mas se observa que na prática este comportamento é desejado. Compiladores, depuradores e sistemas de controle de versão são os principais sistemas que se referem a trechos de código pela linha, gerando várias mensagens para uma versão do programa. O programador então utiliza estas mensagens para modificar o código. Em editores de texto as alterações feitas movem o código, diminuindo a utilidade das mensagens por não conter mais referências a linhas corretas. Em um editor estruturado que mantenha referência às linhas originais, o relatório continua preciso e pode ser utilizado para realizar todas

as alterações necessárias sem consultar o código fonte original ou executar a ferramenta novamente.

Naturalmente esta característica pode confundir o usuário, exigindo uma comunicação clara do editor ou, na pior das hipóteses, serialização completa das estruturas lidas daquele arquivo para reavaliar a linha atualizada e simular o comportamento dos editores de texto.

4.1.2 Dobra de código

Dobra de código se refere ao recurso de omitir as partes internas de uma estrutura, como o bloco de comandos de um if, a fim de economizar espaço na tela. Existem duas situações primárias para este uso: omitir código irrelevante ao que se está editando e "compactar" uma estrutura de alto nível, fornecendo uma espécie de índice ou sumário através das subestruturas dobradas.

Em editores textuais o recurso de dobra de código é limitado por estruturas de fácil identificação, com delimitadores claros ({}, begin/end), e mesmo assim é necessário realizar um parsing simplificado daquela seção a ser dobrada. Por outro lado um editor estruturado pode implementar este recurso de forma trivial, com um simples marcador na estrutura a ser dobrada. A dobra de estruturas arbitrárias também se torna possível, como declaração de literais longos (textos, maps e listas com muitos elementos), abrindo novos usos para o usuário.

4.1.3 Movimentação de longa distância

Alguns modos de navegação são capazes de levar o usuário a locais arbitrários do projeto, como buscas textuais ou sintáticas, e saltos para o local da declaração de um método ou variável. A diferença principal entre estes e os modos de navegação citados nas seções anteriores é que eles podem levar o usuário para locais com maior profundidade na árvore sintática. Uma busca por palavra chave, por exemplo, pode encontrar resultados dentro de uma longa expressão, que está dentro de um bloco de condição com vários caminhos possíveis, dentro de outro bloco de condição, etc.

A navegação para pontos arbitrários é problema por causa da serialização do contexto. Se o próximo nodo for muito longo o editor pode desperdiçar tempo serializando textos que não aparecerão na tela. Já se o nodo anterior for muito longo, há o problema adicional de rolagem para garantir que o nodo alvo se mantenha visível na tela. Este problema é causado pela falta de relação entre a árvore sintática e sua forma serializada. Os nodos possuem liberdade para expandir sua forma serializada de forma arbitrária, trazendo problemas para operações que precisam de acesso aleatório no texto final.

Existem duas soluções principais para este problema. A primeira é serializar os

nodos de forma ingênua, gerando o seu texto completo e recortando as partes necessárias. Esta solução é adequada a editores lidando com códigos simples, já que o próprio programador se preocupa em manter estes nodos com tamanho limitado por questões de legibilidade. Uma segunda solução é manter estimativas de tamanho em cada nodo, similar a solução para navegação por linha. Estes valores permitem análises muito mais precisas, mas exigem constante atualização enquanto o usuário editar o código, além de uma lógica adicional significativa e dependente de linguagem no editor.

Dependendo do tipo do nodo a ser exibido, talvez nenhuma das soluções acima sejam necessárias. Classes e métodos são geralmente considerados estruturas isoladas e poderiam ser serializados como tal. Assim a busca por um método exibiria apenas o método encontrado, independente do tamanho dele e do espaço disponível. Esta alternativa é mais atraente em interfaces modulares (CODE...,) (GRANGER, 2012), pois ela automaticamente extrai apenas o conteúdo importante e permite que o usuário o organize de forma mais eficiente em janelas flutuantes, abas e divisões verticais/horizontais de tela.

4.2 Edição

A edição de código é de longe o problema mais difícil em editores estruturados (OSENKOV, 2007). A organização do código em árvore é bastante elegante para o computador que a está processando, mas pode ser confusa e inconsistente para um usuário humano. Abordagens incluem a simulação de editores textuais (LARCH...,), uma representação gráfica da árvore (SCRATCH...,) e uma abordagem híbrida, mostrando o texto do código mas permitindo a edição direta na árvore (TILED...,). As diferentes abordagens afetam a familiaridade do usuário com a interface, o número e complexidade dos comandos a serem aprendidos e a necessidade de processamento gráfico.

4.2.1 Seleção de estruturas

Para realizar ações o usuário deve selecionar um local e um comando a ser aplicado. Em editores textuais esta operação ocorre através de um cursor que se mantém entre os caracteres do texto, enquanto o usuário utiliza o teclado para selecionar comandos a serem aplicados naquela posição. Em editores estruturados existem duas abordagens principais: manter o cursor entre estruturas (OSENKOV, 2007), ou manter o cursor sobre estruturas (TILED...,).

A maior diferença entre estas duas abordagens são os comandos de seleção e inserção. Na primeira forma, mantendo o cursor entre estruturas, o local da inserção se torna óbvio: ela ocorrerá entre o nodo anterior e o nodo posterior. Mas por não haver nodo selecionado a deleção se torna ambígua: tanto o nodo anterior quanto o posterior se tornam candidatos. Usualmente esta ambiguidade é resolvida com comandos de remoção diferen-

tes, como Backspace versus Delete. Ferramentas de remoção especiais, como Recortar, são limitadas por operarem em apenas uma direção ou exigir uma múltipla seleção.

Já na segunda forma, com o cursor sobre estruturas, o problema oposto ocorre. A remoção se torna óbvia e a inserção ambígua. Alguns editores, mesmo textuais, possuem funcionalidade para inserção "antes" e "depois" do cursor (VIM,), mostrando que uma possível solução é usar o oposto dos comandos da primeira forma: uma única forma de deleção e dois comandos separados para inserção antes e depois do nodo atual. Novamente, ferramentas de inserção especiais, como Colar, ficam limitadas ou exigem dois comandos separados.

4.2.2 Remoção

Em um editor estruturado a operação de remoção é bastante simples por causa da estrutura em árvore. Apenas remove-se o nodo atual e seleciona-se o próximo. Existem alguns casos especiais, mais seus tratamentos são simples:

a) sem próximos nodos no mesmo nível: pode-se selecionar o nodo anterior ou o pai

b) único nodo no nível: é obrigatório selecionar o nodo pai

c) raiz da árvore: impedir a deleção, ou removê-la e recriar uma nova raiz vazia

4.2.3 Inserção

A fácil inserção de estruturas é uma das maiores vantagens dos editores estruturados. Usualmente ela é implementada como uma forma de auto-complete, onde o usuário seleciona o tipo de estrutura que ele deseja inserir a partir de uma lista de possibilidades. Esta é uma interface bastante natural dada a natureza formal da estrutura sendo editada, e extremamente eficiente na proporção entre código escrito por comandos dados.

Um detalhe a ser notado é como inserir estruturas que possuem itens obrigatórios. Normalmente as estruturas são inseridas vazias, maximizando a flexibilidade, mas isto não é possível em todos os casos. Uma atribuição obrigatoriamente deve ter expressões nos seus lados esquerdo e direito, por exemplo. Uma solução é criar subestruturas mínimas, no caso com as expressões mais simples possíveis. O resultado é um editor que insere todas as estruturas com valores nulos, para serem substituídos pelo programador. Como raramente estes são os valores realmente desejados, o editor pode marcar os nodos que foram criados automaticamente para que o usuário possa facilmente selecionar o próximo nodo temporário, a fim de rapidamente substituí-lo.

4.2.4 Movimentação

A movimentação do cursor pode ocorrer através de teclas direcionais ou cliques do mouse. A movimentação com o mouse é bastante intuitiva, selecionando o nodo que gerou o item clicado, seja texto ou gráfico. Uma funcionalidade opcional é a de ampliar a seleção em cliques subsequentes, facilitando a seleção de nodos com poucos itens gerados diretamente. Este é o caso de estruturas em lista, como uma lista de parâmetros, onde apenas as vírgulas entre os valores são criadas diretamente por aquele nodo, dificultando a seleção da estrutura inteira. Com este recurso de ampliação o usuário pode selecionar a lista de parâmetros inteira clicando duas vezes em um dos filhos, similar a seleção de palavra/parágrafo/texto com cliques subsequentes em um editor de texto.

Para movimentação com teclas direcionais a forma mais simples é percorrer a árvore segundo as direções dadas, mapeando movimentos horizontais para nodo próximo/anterior e movimentos verticais para nodo pai/filho. Esta forma é extremamente simples de implementar mas possui sérias limitações de usabilidade: a relação entre a direção que se deseja navegar na árvore e na estrutura serializada podem ser incompatíveis. Por exemplo, um bloco de código costuma ser mostrado na vertical, com cada comando em uma linha. Se o cursor estiver sobre/antes de um destes comandos, será necessário pressionar a tecla para a direita para navegar para o comando que está visualmente abaixo. Para resolver este problema pode-se inverter a direção das teclas, mas isto afetaria as expressões matemáticas, onde os filhos de um nodo são exibidos na horizontal. Até uma abordagem híbrida pode não ser suficiente, dado que algumas estruturas podem misturar as duas formas, como é o caso de declaração de funções. Uma função possui no mínimo um nome, uma lista de parâmetros e um bloco de código associados. Usualmente o nome e a lista de parâmetros são colocados na mesma linha, e o bloco de código abaixo. Neste caso não há uma direção clara para acessar os nodos filhos.

Uma segunda abordagem é serializar a árvore de forma a facilitar esta movimentação. Um exemplo de editor com esta característica é o Scratch (SCRATCH. . . ,), onde a estrutura em árvore tem paralelos claros com aquilo que é mostrado para o usuário. Embora elegante, esta abordagem exigiu uma interface quase que exclusivamente por meio do mouse, com cliques e arrastes. O resultado é uma interface simples e intuitiva, mas pouco eficiente e que não é usada para casos maiores.

4.2.5 Ordenação

Uma operação comum em editores de código é reordenar estruturas, como transpor dois caracteres vizinhos ou mover uma linha de código acima ou abaixo. Esta é uma operação onde corretude sintática e semântica são importantes para evitar bugs, como no caso de mover o incremento de uma variável para antes de sua declaração, ou transpor caracteres inválidos como vírgulas e parênteses.

Editores estruturados possuem tremenda vantagem neste ponto por causa da sua estrutura em árvore. Uma vez que os comandos simples de ordenação, mover para frente e para trás, são implementados, eles se tornam disponíveis para todos os tipos de nodos da estrutura. Em contraste com editores textuais, em um editor estruturado o programador pode reordenar argumentos de uma função, valores dentro de um literal de lista ou de map, métodos ou comandos. Este recurso se torna tão poderoso por ter sido abstraído do contexto de linhas e caracteres, sendo um bom exemplo das vantagens que este paradigma traz.

4.2.6 Seleção de modificadores

Uma funcionalidade opcional para um editor estruturado é tratar modificadores de forma especial. Modificadores podem ser propriedades embutidas no nodo (como atributos em XML, `<person sex="female">`), modificadores de acesso na declaração de uma função (como public e static em "public static myFunction()"), tamanho da lista em uma declaração de array, etc. A diferença entre modificadores e filhos comuns é que os modificadores são mais simples e não possuem filhos próprios. As vantagens em tratá-los de forma diferenciada é removê-los do caminho quando o usuário estiver navegando na árvore, e permitir fácil alteração.

A navegação fica mais simples porque os modificadores são ignorados durante a navegação normal. Isso diminui o número de nodos que precisam ser navegados para que o usuário chegue ao nodo destino. Em nodos com muitos modificadores isto pode ser significativo, como uma função com modificadores de acesso (public), ciclo de vida (static), sincronicidade (synchronized) e possibilidade de ser extendida (final).

O tratamento especial também auxilia na alteração dos modificadores. Comandos especiais podem ser reservados para alternar modificadores binários ou permitir edição de modificadores mais complexos. Esta funcionalidade também tem o bom efeito colateral de diminuir o número de tipos de estruturas, facilitando a configuração do editor para a linguagem alvo.

5 Desvantagens e impedimentos para adoção

Editores estruturados não são uma solução perfeita para os problemas dos programadores. Existem diversas desvantagens e impedimentos para adoção desta tecnologia, a maioria em relação a usabilidade e familiaridade. Dado que é um campo com relativamente poucas pesquisas, também surgem problemas de padronização, tanto no formato dos arquivos utilizados, APIs para acesso programático e interface com o usuário.

5.1 Usabilidade

Problemas de usabilidade são considerados os maiores impedimentos para adoção de editores estruturados (OSENKOV, 2007). Editores textuais exigem do usuário constante cuidado para manter a corretude de seus programas, mas eles podem ser usados com pouquíssimos comandos, além de usarem das letras e símbolos do teclado para representarem os caracteres a serem inseridos. Editores estruturados não possuem estas vantagens, obrigando o usuário a consultar ou decorar uma tabela de possíveis inserções, um problema que é agravado pelo fato desta tabela ser exclusiva para uma linguagem alvo.

A dificuldade de representação da árvore sintática é outro problema que prejudica a usabilidade. Árvores podem ser representadas com elementos gráficos, como um grafo, mas esta abordagem ocupa muito espaço e é de difícil leitura. Exibir a estrutura de uma forma serializada, similar ao texto original, traz benefícios de familiaridade e facilidade de renderização, mas a relação estrutural entre os nodos perde claridade. Alguns editores preferem uma exibição em texto com elementos gráficos no fundo para representar os diferentes níveis da árvore (TILED. . . ,) (OSENKOV, 2007), mas este contraste variável traz problemas de legibilidade.

5.2 Familiaridade

Virtualmente todos os usuários de computador possuem experiência com editores textuais, seja com programas voltados a edição de código fonte, de documentos digitalizados ou campos de entrada na web. A metáfora de “caracteres em uma matriz” utilizada para edição textual tem penetração completa na base de usuários, enquanto a edição de estruturas em árvore ainda é pouco difundida. Esta falta de familiaridade coloca uma grande exigência nos editores estruturados, que além de precisar serem mais eficientes devem ser suficientemente eficientes para vencer a barreira de aprendizagem para serem adotados pelos usuários (OSENKOV, 2007).

A falta de uma metáfora de edição consistente entre diferentes editores estruturados também introduz o problema de migração. Um usuário que optar por usar um editor estruturado raramente será capaz de migrar seu conhecimento para outro editor, mesmo um que também seja estruturado.

5.3 Limitação de linguagem alvo

Editores textuais possui a grande vantagem de serem agnósticos de linguagem, permitindo a edição de programas em linguagens exóticas, muito complexas ou simplesmente não previstas pelo autor do editor. É importante notar que isto inclui linguagens naturais, já que uma vez o usuário estando familiarizado com o editor ele pode preferí-lo para edição de todas as formas de texto.

Editores estruturais exigem no mínimo um parser para a linguagem alvo, com a qualidade da árvore gerada pelo parser refletindo diretamente na qualidade de edição da árvore, medida por profundidade e correspondência lógica com o formato serializado. O resultado é que muitos editores estruturados são compatíveis com apenas uma linguagem, possivelmente de uso exclusivo com aquele editor, por dificuldades de implementação. Limitações como esta exigem que o usuário tenha um editor secundário, completamente textual, para lidar com casos onde um arquivo de texto precisa ser editado sem suporte do editor.

5.4 Dificuldades de implementação

Editores estruturados são inerentemente mais complexos de criar do que editores textuais simples por causa do passo de parsing e tratamento específico de linguagem. Um editor textual pode ser criado com um simples campo de texto, alguns poucos comandos de edição e funções de acesso a arquivos, o que é ainda mais facilitado com as primitivas de edição de texto presentes em quase todos os frameworks GUI. Em contraste, editores estruturados exigem um parser completo da linguagem alvo (para processamento dos arquivos fonte), uma forma de serializar a árvore sintática para exibição e escrita em disco, restrições específicas nos nodos da árvore para garantir a corretudade sintática, além dos comandos de edição.

A complexidade de editores estruturados é menor do que IDEs textuais, com seus analisadores de código rodando em segundo plano (OSENKOV, 2007), mas o custo inicial de implementação é muito maior. Enquanto IDEs podem nascer de editores textuais simples, editores estruturais tem uma barreira inicial muito maior para serem funcionalmente úteis. Esta barreira diminui o número de alternativas disponíveis para o usuário e dificulta pesquisas na área.

5.5 Dificuldades de padronização

Editores estruturados poderiam fazer bom uso de padrões específicos para suas necessidades, como uma metáfora de edição unificada e formato de arquivo para árvores sintáticas. Atualmente cada editor implementa suas próprias soluções para estes problemas, desperdiçando tempo da comunidade e fragmentando o ambiente. Uma metáfora de edição, correspondente a metáfora de "caracteres em uma matriz" utilizada por editores textuais, facilitaria a adoção por diminuir a barreira de aprendizagem para novos editores. Já um formato padronizado para armazenamento de árvores semânticas poderia funcionar como uma espécie de cache global para os arquivos fontes que já foram analisados, diminuindo o custo computacional de abrir um projeto grande. Tal formato também permitiria a criação de programas especializados em parsing de código fonte para geração destes arquivos, tirando a necessidade de implementação de um parser completo em cada editor.

Parte II

Implementação

6 Criação de um Editor Estruturado

Para melhor explorar o tópico foi implementado um editor de código completamente estruturado (Figura 11), utilizando algumas das funcionalidades previstas nas seções anteriores. O objetivo deste editor era testar na prática as vantagens e desvantagens de um editor estruturado, permitir uma comparação mais objetiva com editores convencionais, e tentar vencer alguns dos desafios que esta classe de editores possui.

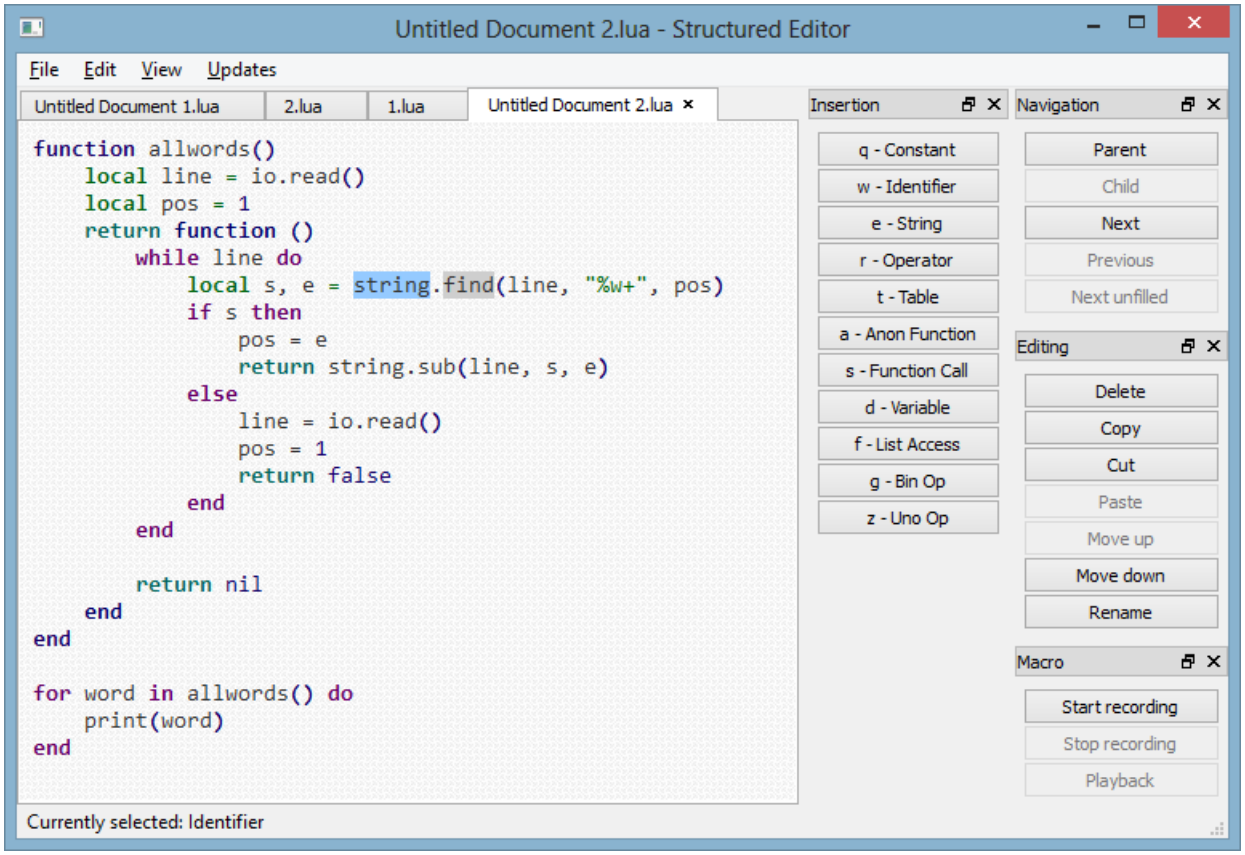

Figura 11 – Aparência do editor criado, em seu estado atual

Este editor foi criado pensando em um uso diário, para linguagens de programação de propósito geral, trabalhando em conjunto com outros programadores que podem estar usando editores textuais. Para atingir este nível de praticidade foram feitas diversas escolhas importantes, como exibir o código em edição de forma textual, isolar o código de parsing e as estruturas do editor em si, e extrair detalhes de funcionamento como formato de exibição e teclas de atalho para arquivos de configuração.

7 Tecnologias

A linguagem Lua (THE...,) foi escolhida como linguagem primária do editor (Figura 12). Por ser uma linguagem de script, bastante usada para configuração de aplicações maiores, existem várias ferramentas dedicadas a seu parsing, de gramáticas a bibliotecas para outras linguagens. Sua construção também é bastante simplificada, tendo um número pequeno de regras sintáticas já planejadas para facilitar a sua interpretação. Apesar de sua simplicidade a linguagem é bastante expressiva, com suporte a metaprogramação, e similar a linguagens maiores como Python, Ruby e Javascript.

```
function factorial(n)
    if n == 0 then
        return 1
    else
        return n * factorial(n - 1)
    end
end
```

Figura 12 – Exemplo de código lua, mostrando uma função fatorial implementada recursivamente

A linguagem escolhida para construção do editor foi Python (PYTHON...,), por questões de eficiência de programação e familiaridade. Esta escolha foi bastante pessoal e teoricamente qualquer outra linguagem moderna poderia ter sido utilizada. Houveram problemas de desempenho na parte do parser, mas estas seções mais críticas podem ser extraídas do programa principal e reimplementadas em outras linguagens facilmente.

O parsing de programas Lua foi feito com a biblioteca PyParsing (PYPARSING,) e uma gramática adaptada da gramática disponível na documentação oficial da linguagem Lua. Esta biblioteca permite que a gramática seja descrita em código Python regular e possui diversas funcionalidades para facilitar o parsing de linguagens de programação, como produções de strings e números, tratamento automático de espaços em branco e agrupamento de produções em classes fornecidas pelo usuário da biblioteca.

Para criação da interface gráfica foi usado o framework PyQt (RIVERBANK...,), uma sistema capaz de fazer a comunicação entre programas Python e o framework Qt (QT...,) original em C++. Assim esta ferramenta permite o uso de todas as funcionalidades do framework Qt, como portabilidade, extensibilidade e as centenas de componentes interativos.

A exibição do código em edição é feita através de um componente QWebView, capaz de exibir páginas web. Isto permitiu que toda a formatação do código seja feita em

HTML/CSS, facilitando a implementação e permitindo uma customização mais simples pelo usuário.

Por fim, versões executáveis para o sistema operacional Windows foram geradas utilizando a ferramenta py2exe (PY2EXE,). Esta ferramenta identifica as dependências necessárias de um programa em Python e as extrai para um arquivo compactado, junto com um interpretador portável. Isto permite que o programa final seja executado em qualquer máquina independente da disponibilidade de interpretadores Python ou das bibliotecas utilizadas.

8 Arquitetura

O editor foi implementado em três módulos distintos: ast, core e gui (Figura 13).

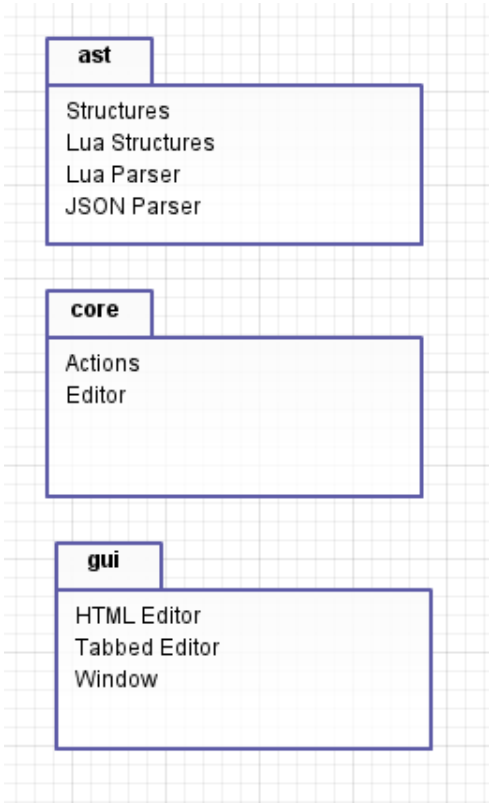

Figura 13 – Divisão do projeto em módulos

a) *ast* é o módulo que contém as informações das linguagens alvo e seus parsers. Este módulo é responsável por transformar o texto do código em sua árvore sintática e vice versa. A árvore sintática é representada por classes que herdam da classe *Node*, onde cada uma destas classes representa uma estrutura sintática da linguagem.

b) *core* contém o núcleo do editor, com suas operações. Aqui estão definidas as ações que um usuário é capaz de executar sobre a árvore sintática do código. As ações são representadas por classes que herdam da classe *Action*, a fim de permitir uma interface única para execução, além de funcionalidades de desfazer e refazer.

c) *gui* engloba o código da interface gráfica. Isto inclui a janela principal, *Window*; as barras de ferramentas *MacroWindow*, *InsertionWindow*, *EditWindow* e *NavigationWindow*; e os componentes para exibição gráfica do código, *HtmlEditor* e *TabbedEditor*.

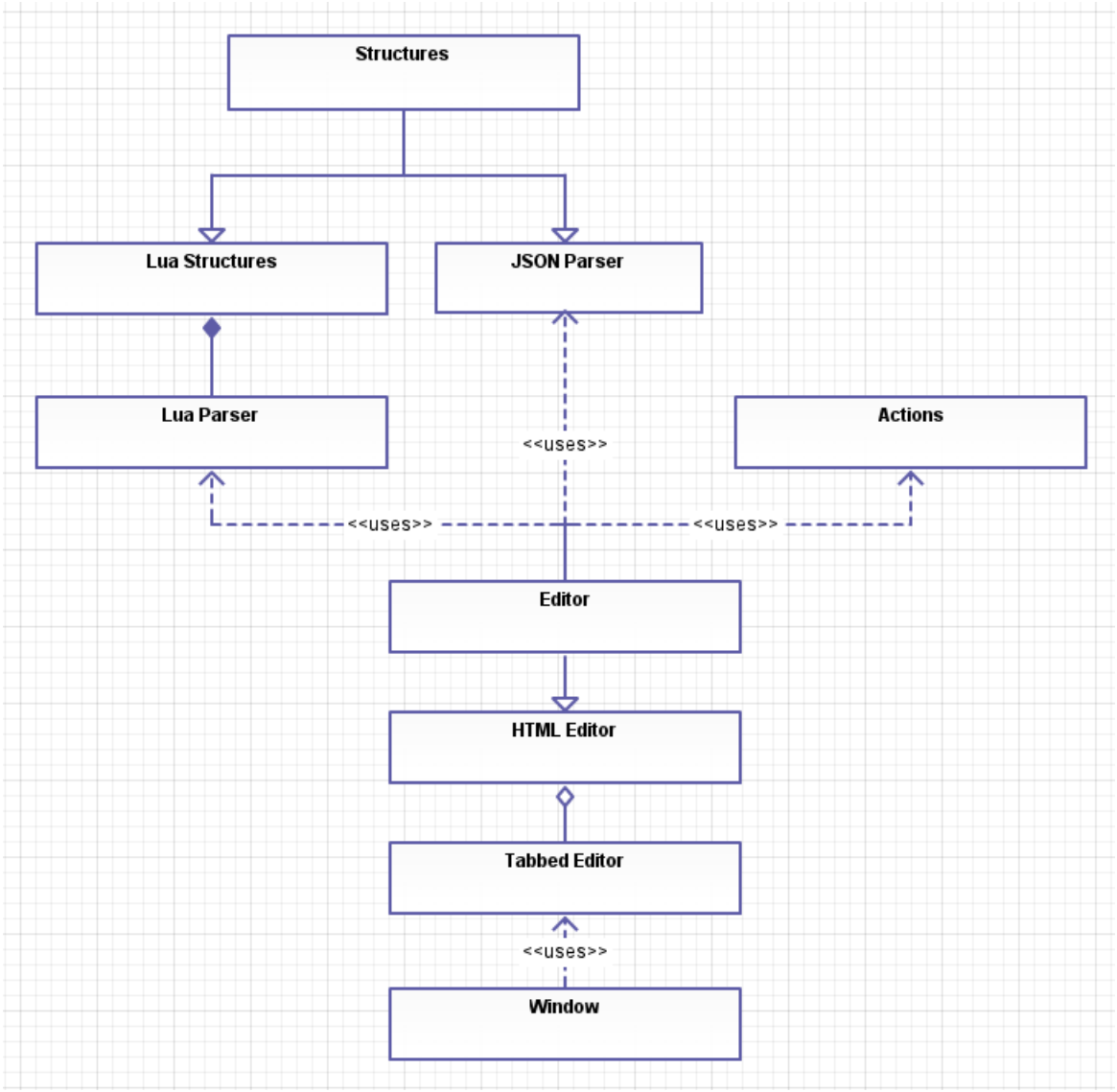

Figura 14 – Diagrama abstrato dos diferentes pacotes

Na Figura 14 é possível ver a estrutura dos pacotes do projeto. Apesar de não representarem classes diretamente, a maioria destes pacotes tem conteúdo homogêneo e foi esta relação de alto nível que foi representada. Lua *Structures*, por exemplo, contém diversas classes, sendo que todas estendem as classes de *Structures*.

8.1 Funcionalidades

O editor atualmente conta com diversos recursos para facilitar o uso no dia a dia, como barras de ferramentas destacáveis com os comandos de edição, atualizações automáticas, comandos de desfazer e refazer, e uma opção para restabelecer as características "de fábrica". Embora importantes para o usuário final, o objetivo deste trabalho é analisar o desempenho deste editor em relação a suas características estruturadas, portando as pró-

ximas seções se limitam a funcionalidades que sejam exclusivas de editores estruturados ou de difícil replicação.

8.1.1 Formatação Automática

O texto exibido para o usuário é uma serialização direta da árvore sintática e sua formatação não tem relação com a formatação do arquivo original. Isto significa que todo código exibido está automaticamente formatado de acordo com as preferências do usuário (mais informações sobre isto na próxima seção).

Apesar de ser um recurso relativamente simples, a formatação incorreta confunde o leitor e pode levar a erros. Considere o código condicional a seguir:

```
if i == 0 then
    a = b
    c = d
e = f
end
```

Uma leitura distraída do código acima pode levar o programador a pensar que o comando "e = f" é sempre executado, independente da condição "i == 0". Isto não é verdade, uma vez que este comando ainda se encontra dentro do bloco da condição. Em outras linguagens existem construções ainda mais ambíguas, como o bloco de condição:

```
if (i == 0)
    a = b
    c = d
e = f
```

Este é um trecho de código válido em Java, C e C++ onde a formatação sugere que o comando "c = d" faz parte dos comandos condicionais, quando apenas o primeiro comando, "a = b", está sendo controlado.

A formatação automática deste editor impede que problemas deste tipo ocorram tanto na exibição do código quando no programa salvo em disco.

8.1.2 Separação de formato de exibição e de saída

As formas que o código é exibido ao usuário e que ele é salvo em disco são controladas por arquivos de configuração distintos. O arquivo "theme.ini" contém, entre outras configurações, informações de como serializar a árvore sintática. Estas informações estão no formato de templates, como por exemplo:

```
namedfuncion = function {name}({parameters}){body}\nend
```

Nestes templates os trechos a serem substituídos pelos valores reais estão entre chaves {}, enquanto as outras palavras são valores literais. Neste caso o template informa ao programa que uma declaração de função com nome (namedfunction) deve ser exibida com o prefixo "function", seguido pelo nome da função, a lista de parâmetros entre parênteses, o corpo da função, e um sufixo "end" em uma nova linha.

O usuário tem total controle para modificar os valores literais, reposicionar os campos a serem substituídos e até mesmo omitir informações. Para realizar uma revisão rápida de uma API, por exemplo, o usuário pode optar por omitir o corpo das declarações de funções e o sufixo "end", permitindo que mais métodos sejam visíveis.

A configuração de formato de saída é dada da mesma forma, mas em um arquivo separado, chamado "output_format.ini". A decisão desta separação foi em vista de equipes de programadores que queiram utilizar uma convenção única para os arquivos salvos, mas manter a exibição a escolha de cada um. Assim os integrantes desta equipe podem compartilhar os seus arquivos "output_format.ini" enquanto personalizam o arquivo "theme.ini" a seu gosto.

8.1.3 Realce de sintaxe com garantia de corretude

O realce de sintaxe, também conhecido como *Syntax Highlight*, é habilidade de exibir estruturas em cores e fontes próprias, facilitando a leitura. É um recurso quase universal dos editores de código, mas a sua implementação usualmente é através de expressões regulares. O resultado é que existem muitos falsos positivos, estruturas que são coloridas de forma incorreta. Este editor estruturado insere as informações de realce durante a fase de serializar e portanto garante a sua corretude. Além disso as configurações de realce podem ser personalizadas no mesmo arquivo "theme.ini", onde o usuário pode definir estilos *CSS* para cada tipo de estrutura.

8.1.4 Movimentação entre estruturas

Toda movimentação do cursor é feita diretamente na árvore sintática. Quando uma estrutura estiver selecionada, o comando de movimentação para baixo seleciona a próxima estrutura no mesmo nível, enquanto os comandos laterais selecionam os filhos ou os pais da estrutura atual. Esta funcionalidade é bastante útil para navegação de grandes estruturas, como métodos ou condicionais com muitos comandos.

Diferente do cursor convencional, o cursor do editor criado funciona como uma seleção permanente. Ele representa o nodo atualmente em foco e portanto todo o conteúdo do nodo é exibido com aparência de seleção.

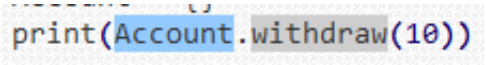

Figura 15 – Exemplo de seleção de nodo

Para explicitar a relação do nodo atual com os nodos ao redor, optou-se por também exibir os nodos no mesmo nível de forma diferenciada. Na Figura 15) é possível observar que os nodos “Account” e “withdraw” estão no mesmo nível, com a seleção principal em azul e a secundária em cinza, e portando uma navegação lateral irá alternar entre os dois.

8.1.5 Seleção de estruturas com o mouse

A movimentação direcional é bastante útil para navegar grandes estruturas, mas foi notado que ela é ineficiente e pouco intuitiva para estruturas menores, como expressões matemáticas. A fim de sanar esta deficiência foi implementado o recurso de seleção de estruturas com o mouse, onde o usuário pode clicar na estrutura para onde ele deseja navegar.

Naturalmente o texto visível para o usuário é apenas uma serialização da árvore sintática, e é necessária uma forma de mapear o trecho selecionado para o nodo correspondente. Isto foi feito com anotações no código HTML gerado, documentando o nodo responsável pela geração de cada trecho do texto.

Algumas estruturas geram poucos textos próprios, como a estrutura de lista de parâmetros. Esta estrutura contém vários identificadores como nodos fillhos, que são serializados de forma recursiva, mas o único texto gerado diretamente são as vírgulas entre estes nodos. Com uma implementação ingênua a única forma do usuário selecionar o nodo da lista de parâmetros seria clicando na vírgula (Figura 16).

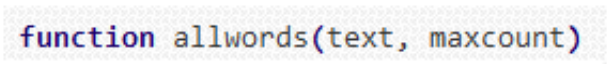

Figura 16 – Exemplo de uma estrutura onde apenas a vírgula foi gerada diretamente

Para solucionar este problema foi implementada uma função de ampliação de seleção, similar aos editores textuais. Ao clicar em um nodo a primeira vez, apenas ele é selecionado. Ao clicar novamente em um curto intervalo de tempo, o nodo pai é selecionado, e assim por diante. Este recurso é equivalente a seleção de letra/palavra/parágrafo/texto encontrado em muitos editores convencionais, porém mais abstraído por ser capaz de navegar tantos níveis quanto existirem na árvore sintática.

8.1.6 Inserção de estruturas completas

Ao selecionar um nodo da árvore, a barra de ferramentas “Insertion” (Figura 17) é atualizada para exibir os tipos de estruturas disponíveis para aquele nodo. Cada alternativa é exibida como um botão e também é mapeada a uma tecla do teclado.

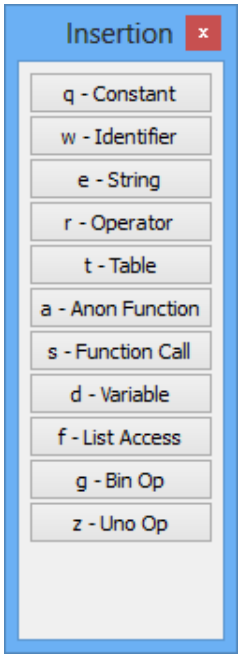

Figura 17 – Barra de ferramentas de inserção mostrando as opções de estruturas do tipo Expression

Ao pressionar o botão ou a tecla de atalho correspondente, o editor automaticamente insere uma estrutura daquele tipo no local atualmente selecionado. Como esta inserção ocorre na árvore sintática, uma única operação de inserção é capaz de inserir dezenas de caracteres ao código fonte. A ordem das teclas de atalho dão preferência as letras do lado esquerdo do teclado, permitindo que o usuário realize inserções mantendo a mão direita no mouse.

No caso de estruturas compostas, como blocos condicionais, o editor gera valores padrões onde necessário. No caso de estruturas que exigem um nodo do tipo Expression, este valor é um identificador com o nome “value”; para operadores matemáticos é o sinal “+”; etc. Isto permite que o programador rapidamente gere um esqueleto do seu programa, com as estruturas de alto nível, para então definir os detalhes. Para facilitar este fluxo de trabalho existe uma operação de navegação “Next unfilled”, que seleciona o próximo nodo de valor padrão que necessita de atenção do usuário.

É importante notar que as opções de estruturas são extraídas dos objetos da árvore sintática, mantendo a implementação do editor livre de qualquer conhecimento da linguagem alvo.

8.1.7 Operações de reordenação

Alguns editores de texto possuem operações para transpor linhas e caracteres. Estes comandos permitem a reordenação de textos já inseridos, mas são limitadas em escopo por trabalharem apenas nestes dois níveis, linhas e caracteres.

O editor implementado possui dois comandos de reordenação que operam diretamente na árvore sintática, nomeados "Move up" e "Move down" (Figura 18). Eles alteram a relação do nodo selecionado com os nodos do mesmo nível, trocando-o de lugar com o próximo nodo ou o nodo anterior.

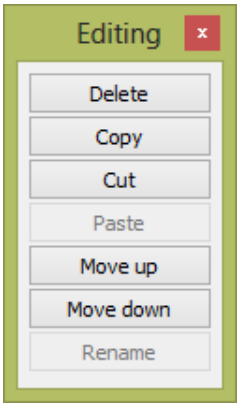

Figura 18 – Barra de ferramentas de comandos de edição genéricos

Com estas operações o usuário é capaz de facilmente reordenar comandos em qualquer tipo de bloco, parâmetros de um método, declaração de itens em uma tabela, etc.

Esta simples operação permite que qualquer tipo de estrutura seja reordenada, independente do tamanho, complexidade e mais importante, linguagem. Esta operação, assim como todas as operações de edição, atua diretamente na árvore sintática sem conhecimento das estruturas representadas, o que torna as operações genéricas para qualquer linguagem que o editor seja capaz de realizar o parse.

8.1.8 Interação com códigos em texto

Não é realista esperar que todo código que o usuário encontre esteja corretamente escrito em arquivos corretamente nomeados e de fácil acesso em seu computador, e que os programas escritos nunca precisem ser convertidos novamente em texto. Na prática o usuário terá necessidade de carregar programas a partir de textos isolados, como soluções em sites de ajuda, e também exportar o código escrito para meios que suportam apenas texto. Com esta necessidade em mente foram criados dois recursos: o menu "Parse text" e a operação "Copy".

O menu “Parse Text” fornece uma janela de edição textual para o usuário entrar com programas isolados (Figura 19).

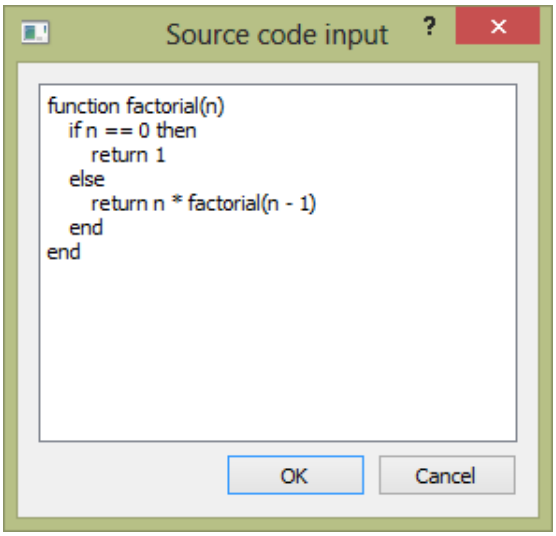

Figura 19 – Exemplo de entrada na janela acessível pelo menu Parse Text

O objetivo desta função é permitir que o usuário edite códigos encontrados isoladamente, como em sites de ajuda, documentação ou emails. Ao entrar com o código o editor abre uma nova aba para edição deste programa. Após ser lido corretamente, o programa é tratado pelo editor exatamente como um programa lido de um arquivo convencional ou criado a partir de uma raiz vazia.

Para a conversão entre a árvore sintática e texto o usuário pode utilizar a operação “Copy”. Esta operação tem duas funções: copiar o nodo atual para uma área de trabalho interna do programa, permitindo que o nodo seja replicado em outras seções; e salvar uma cópia da estrutura selecionada, em texto, para a área de trabalho do sistema operacional. Esta cópia em texto salva na área de trabalho permite que o usuário copie estruturas de dentro do editor e cole normalmente em aplicações textuais, mantendo até a formatação HTML em editores que a suportam.

8.1.9 Filtro de entradas

A operação “Rename” é capaz de modificar os valores literais de estruturas, como o nome de um identificador, o símbolo de um operador matemático ou o conteúdo de uma string (Figura 20). Estes são valores puramente textuais e não possuem representação estruturada, então a única forma de modificá-los é através de um campo de texto convencional. A diferença entre modificar o valor de uma string em um editor textual e em um editor estruturado é que o segundo está ciente da operação que está ocorrendo e pode auxiliar o usuário.

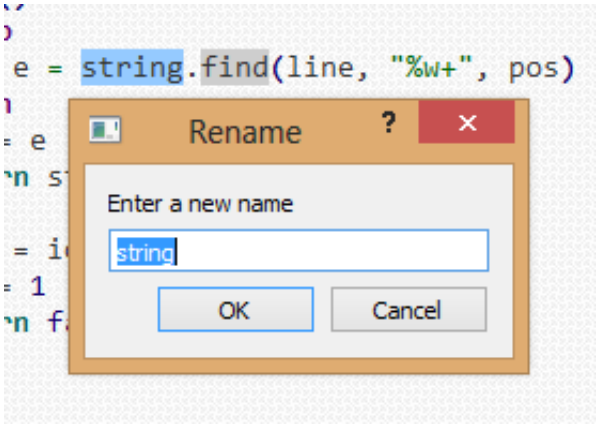

Figura 20 – Renomeando um identificador

A operação “Rename” é capaz de modificar o input recebido para evitar problemas sintáticos. Um nome de variável, por exemplo, não deve conter espaços e portanto o “Rename” automaticamente substitui estes espaços por underscore. Isto permite que o usuário utilize a tecla espaço para separar as palavras no nome da variável, ao invés do underscore, ou até mesmo copie o conjunto de palavras de outro local.

Outra forma que esta operação auxilia a entrada é na mudança de strings literais. Estas estruturas são delimitadas por aspas, e portanto qualquer ocorrência deste caractere especial deve ser tratada. A operação “Rename” realiza esta mudança automaticamente, possibilitando que o usuário digite qualquer tipo de texto naquele campo sem preocupação com tratamento destes caracteres.

Por fim, o input é validado de acordo com as regras léxicas da linguagem. Cada estrutura literal (nome de variável, string, operador matemático) possui uma expressão regular que representa os inputs aceitáveis para aquele local. A operação de “Rename” verifica se o input está conforme a regra léxica, garantindo a corretude daquela estrutura.

8.1.10 Inserção automática de parênteses

A serialização de expressões matemáticas é um problema quando se considera a precedência de operadores. Embora na árvore sintática não haja dúvidas da forma que as expressões são agrupadas, a sua forma textual não é tão clara. A fim de sanar este problema todas as expressões matemáticas recebem parênteses automaticamente conforme necessário (Figura 21).

```
value = (value - value) / (value + value)
```

Figura 21 – Exemplo de código com inserção automática de parênteses

Este recurso permite que o usuário tenha sempre certeza da forma que a expressão será avaliada pelo compilador ou interpretador da linguagem. Outra possível solução é utilizar operadores pré-fixados.

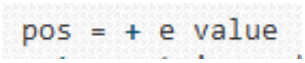

Figura 22 – Exemplo de código com operator pré-fixado

A colocação do operador permite visualizar a ordem de precedência sem parênteses, com o sacrifício da familiaridade (Figura 22). É interessante notar que esta solução pode ser implementada apenas alterando o template de serialização da estrutura de operador binário:

Template com operador infixado:

```
binop = {left_side} {operator} {right_side}
```

Template com o operador pré-fixado:

```
binop = {operator} {left_side} {right_side}
```

Esta modificação pode ser feita no arquivo de tema "theme.ini" e é independente da forma que a expressão será salva em disco (definido em "output_format.ini"). Operadores pré-fixados não são aceitos pela gramática da linguagem Lua, mas não há problema algum em utilizá-los para melhorar a leitura do programa se o usuário assim desejar.

8.1.11 Operações de alto nível

As operações disponíveis neste editor são de mais alto nível que em editores textuais, modificando a estrutura do código diretamente. Este fato permite que o editor tenha melhor entendimento da intenção do usuário e que funções de fazer/refazer e macros sejam mais úteis. Por exemplo, o usuário é capaz de desfazer operações como "declarar nova função" em único passo ao invés de desfazer todas as operações em "inserir caracteres f, u, n, c, t, i, o, n...".

As ferramentas de Macro (Figura 23) também se beneficiam desta visão de mais alto nível. O objetivo destas ferramentas é gravar as ações do usuário em um certo intervalo de tempo para serem repetidas posteriormente.

Em editores convencionais as ferramentas de macro gravam operações orientadas a caractere, e portanto tem problemas de lidar com estruturas onde o número de caracteres varia, como funções com nomes diferentes. Neste editor elas gravam operações como

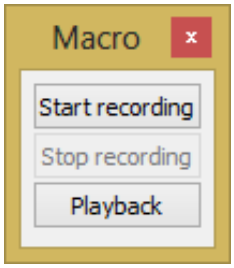

Figura 23 – Barra de ferramentas para controle de macros

"Selecionar próxima estrutura" e "Trocar estrutura de lugar com a estrutura anterior", que são independentes do texto. Isto permite que a repetição das ações tenham maiores chances de realizar as ações que o usuário desejava.

Esta diferença de visão de alto nível, além de permitir que o usuário controle o histórico de ações mais facilmente, pode dar origem a ferramentas com melhor entendimento do processo de edição. Um sistema de controle de versão, por exemplo, pode gravar as alterações em termos de operações de alto nível como as que foram aqui citadas. Com estas informações tal sistema seria capaz de realizar fusões (merge) de ramificações com maior qualidade, responder questões como "esta função foi escrita aqui ou movida de outro lugar" e fornecer resumos de mais alto nível das modificações realizadas.

8.1.12 Suporte a outras linguagens

O objetivo deste editor não era apenas editar programas Lua, mas sim dar suporte a qualquer linguagem. Por este motivo foi adicionado suporte a mais uma linguagem, JSON (Figura 24). Embora não seja uma lniguagem de programação, e sim para transferência de dados, sua natureza simples e estruturada a torna uma boa candidata para um editor estruturado.

O comportamento do editor com esta linguagem é exatamente igual a Lua, com suporte as mesma operações e funcionalidades mencionados anteriormente. Isto se deve ao fato do editor ser agnóstico de linguagem, editando a árvore sintática de forma genérica.

8.2 Limitações

Naturalmente o editor desenvolvido possui diversas limitações, algumas por culpa da implementação e outras que consideradas intrínsecas da categoria de editores estruturados. Dado o escopo do projeto também não foi possível assegurar o funcionamento correto de todas as funções, e portanto uso diário não é recomendado. Apesar disso o editor é bastante flexível e genérico para receber extensões para outros recursos ou linguagens diferentes.

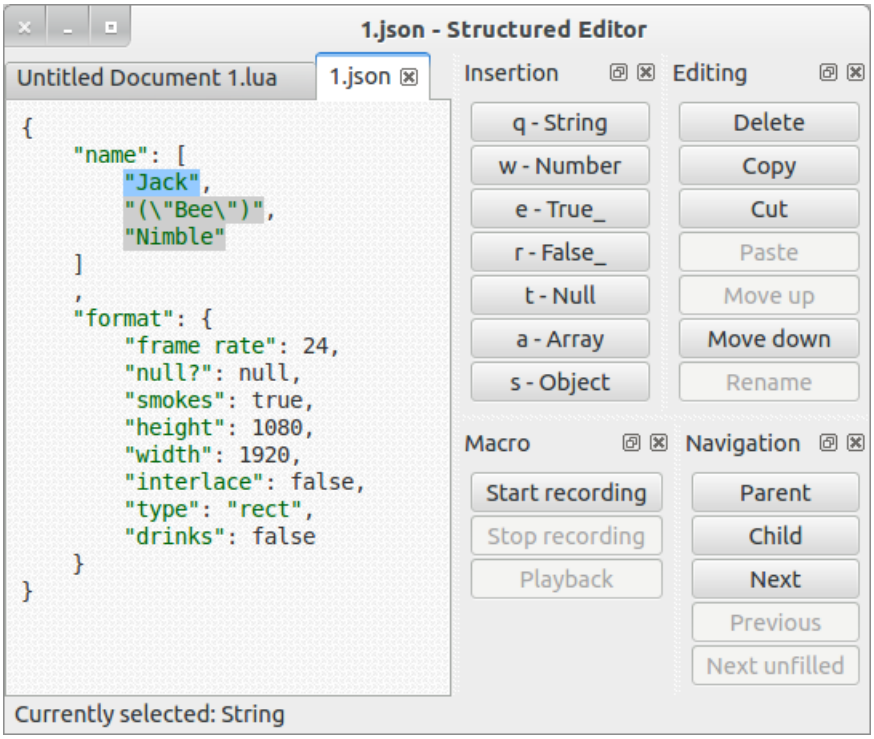

Figura 24 – Edição de um arquivo no formato JSON

8.2.1 Inserção

Toda inserção deve ocorrer utilizando a barra de ferramentas de inserção. Embora as opções lá listadas tenham teclas de atalho simples, existem mais de 30 estruturas Lua disponíveis ao todo. Apesar de listar apenas as opções válidas naquela posição, ainda são necessários mais de 10 teclas de atalho simultâneas no caso de inserção de expressões e comandos.

O resultado é que o grande gargalo de performance dos novos usuários é na localização da estrutura desejada e é necessária muita prática para conseguir memorizar as estruturas principais. Este problema é natural dos editores estruturados, mas talvez possa ser amenizado com inserções por nome, digitando o início do nome da estrutura para inserí-la, ou com uma opção de menu mais intuitiva.

8.2.2 Navegação

A navegação do cursor em grandes estruturas está bastante eficiente, mas expressões pequenas com vários elementos ainda são um problema para o usuário. A seleção com o mouse é capaz de resolver o problema, mas a movimentação da mão do teclado para o mouse é um movimento lento e indesejado pelos usuários. Uma forma alternativa e mais eficiente de navegação para estas pequenas estruturas melhoraria bastante a usabilidade.

Outro problema é que navegação direcional é pouco intuitiva quando a direção

das estruturas muda. Um bloco de comandos, por exemplo, é uma estrutura onde os nodos filhos são organizados em uma lista vertical e o uso das setas cima/baixo é natural para sua navegação. Mas nos próprios comandos a estrutura é horizontal, usualmente ocupando apenas uma linha. Nestes casos a navegação entre filhos continua utilizando as setas cima/baixo, o que é bastante confuso para o usuário. A solução óbvia de mudar a direção de movimento de acordo com a estrutura selecionada não é adequada por culpa de estruturas que utilizam as duas direções, como uma declaração de função: o nome e a lista de parâmetros estão alinhados horizontalmente, enquanto o corpo da função está posicionado verticalmente destes dois.

8.2.3 Falta de modificações sintaticamente inválidas

Por ser um editor completamente estruturado, o programa exige que o código em edição esteja sintaticamente válido durante todas as modificações. Esta característica pode ser indesejada em alguns casos onde a diferença textual é pequena, mas a diferença sintática é enorme. Transformar um bloco de condição If em um bloco de repetição While em um editor textual é uma modificação pequena, alterando apenas os caracteres da palavra chave. Já em um editor estruturado é necessário substituir a estrutura por inteiro.

Atualmente o usuário é obrigado a recriar a estrutura e copiar os nodos filhos desejados. Uma possível alternativa é adicionar uma operação de substituição, onde uma estrutura é substituída por outra tentando manter o conteúdo dos filhos intactos. Esta opção não foi estudada a fundo, então podem haver problemas inesperados com esta abordagem.

8.2.4 Desempenho do parser

Devido a escolha de linguagem do editor (Python) e a biblioteca de parsing (PyParsing), o desempenho do parser está inadequado a códigos complexos. Os exemplos mostrados até aqui são analisados instantaneamente, mas programas maiores, com centenas de linhas, levam vários segundos para serem carregados.

Uma possível solução é extrair o código de parsing para um programa externo, em uma linguagem e biblioteca com melhor desempenho. Tal programa pode se comunicar com o editor através de formatos padronizados, como JSON ou XML, para representar a árvore sintática analisada.

8.2.5 Limitação de linguagem

O editor naturalmente precisa ter conhecimento da linguagem alvo para realizar sua edição, ou ao menos fazer o parse de sua árvore sintática. O resultado desta carac-

terística é que a gama de linguagens passíveis de edição é extremamente limitada e sua ampliação custosa.

Além disso há o problema de que várias linguagens em uso não são facilmente analisadas, por complicações sintáticas ou uso de macros. C++ é um bom exemplo deste problema, onde o sistema de templates é Turing completo e portanto seu parsing é um problema indecidível, e além disso existem ambiguidades naturais da própria gramática. Linguagens deste tipo dificilmente serão portadas para editores estruturados, onde o uso de editores híbridos pode ser mais adequado.

Conclusão

A edição de código em modo estruturado possui diversas vantagens em relação a edição puramente textual, e provavelmente muitas outras ainda serão encontradas. Programadores utilizando editores estruturados ou, em menor grau, híbridos, podem esperar uma edição mais simples, com menor taxa de erros e facilmente estensível para suas necessidades. Aliado com o aumento de demanda de performance, isto significa que a necessidade de editores estruturados vem aumentado, além de sua implementação se tornando mais fácil. Editores textuais vem adotando recursos de editores híbridos e as próprias linguagens e ferramentas se adaptando para facilitar o processamento estruturado, para edição ou não.

Apesar desta necessidade, os esforços nesta área ainda são modestos e o campo de editores para linguagens de propósito geral se encontra completamente aberto. É de se esperar que novas soluções surjam nesta área nos próximos anos, trazendo a tona não apenas as funcionalidades aqui mencionadas, mas também novos recursos, ainda não previstos, para facilitar a vida do programador. A adoção provavelmente será lenta, considerando o investimento necessário para adquirir familiaridade com este paradigma.

Naturalmente existem problemas nesta área. Foram apontados vários problemas da edição estruturada, de questões temporárias como problemas de familiaridade, a problemas mais profundos como a inabilidade de suportar operações sintaticamente inválidas.

Por fim, o editor proposto atendeu as expectativas, demonstrando as qualidades desejadas e exemplificando as funcionalidades possíveis. Embora ainda seja um protótipo, este editor está completo o bastante para escrita e edição de programas na linguagem alvo e pode ser estendido para sanar os defeitos encontrados e implantar novas funcionalidades.

Referências

ATWOOD, J. *Death to the Space Infidels!* 2009. Website. Disponível em: <http://www.codinghorror.com/blog/2009/04/death-to-the-space-infidels.html>. Acesso em: 23.3.2013. Citado na página 26.

CODE Bubbles Project: Rethinking the User Interface Paradigm of Integrated Development Environments. Website. Disponível em: <http://www.andrewbragdon.com/codebubbles_site.asp>. Acesso em: 17.7.2013. Citado na página 45.

GRANGER, C. *Light Table – a new IDE concept.* 2012. Website. Disponível em: <http://www.chris-granger.com/2012/04/12/light-table—a-new-ide-concept/>. Acesso em: 17.7.2013. Citado 3 vezes nas páginas 19, 23 e 45.

HERKEN, R. *The universal Turing machine: a half-century survey.* [S.l.]: Springer, 1995. Citado na página 15.

HOPCROFT, J. F. *Introdução à Teoria de Autômatos, Linguagens e Computação.* [S.l.]: Elsevier, segunda edição, 2003. Citado na página 15.

LARCH Environment. Website. Disponível em: <http://larch.pythonanywhere.com/>. Acesso em: 17.7.2013. Citado 2 vezes nas páginas 21 e 45.

OSENKOV, K. *Designing, implementing and integrating a structured C# code editor.* Dissertação (Dissertação de Doutorado) — Brandenburg University of Technology, 2007. Citado 4 vezes nas páginas 20, 45, 49 e 50.

PY2EXE. Website. Disponível em: <http://www.py2exe.org/>. Acesso em: 17.7.2013. Citado na página 58.

PYPARSING. Website. Disponível em: <http://pyparsing.wikispaces.com/>. Acesso em: 17.7.2013. Citado na página 57.

PYTHON Programming Language. Website. Disponível em: <http://www.python.org>. Acesso em: 17.7.2013. Citado 2 vezes nas páginas 25 e 57.

QT Project. Website. Disponível em: <https://qt-project.org/>. Acesso em: 17.7.2013. Citado na página 57.

RIVERBANK | Software | PyQt. Website. Disponível em: <http://www.riverbankcomputing.com/software/pyqt/intro>. Acesso em: 17.7.2013. Citado na página 57.

SCRATCH – Imagine, Program, Share. Website. Disponível em: <http://scratch.mit.edu/>. Acesso em: 17.7.2013. Citado 2 vezes nas páginas 45 e 47.

SMALLTALK. Website. Disponível em: <http://www.smalltalk.org/main/>. Acesso em: 17.7.2013. Citado na página 19.

THE Programming Language Lua. Website. Disponível em: <http://www.lua.org>. Acesso em: 17.7.2013. Citado na página 57.

TILED Text Engine. Website. Disponível em: <http://www.tiledtext.com/>. Acesso em: 17.7.2013. Citado 2 vezes nas páginas 45 e 49.

VIM. Website. Disponível em: <http://www.vim.org/>. Acesso em: 17.7.2013. Citado na página 46.